2	<< >> (for manifest arrays).
1	; (semicolon as separator between an Assertion_clause and the next)

~~G.4~~

G.2 RESERVED WORDS

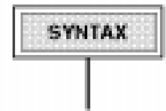
alias	**all**	**and**	**as**	*BIT*	*BOOLEAN*
CHARACTER	**check**	**class**	**creation**	*Current*	**debug**
deferred	**do**	*DOUBLE*	**else**	**elseif**	**end**
ensure	**expanded**	**export**	**external**	**false**	**feature**
from	**frozen**	**if**	**implies**	**indexing**	**infix**
inherit	**inspect**	*INTEGER*	**invariant**	**is**	**like**
local	**loop**	*NONE*	**not**	**obsolete**	**old**
once	**or**	*POINTER*	**prefix**	*REAL*	**redefine**
rename	**require**	**rescue**	*Result*	**retry**	**select**
separate	*STRING*	**strip**	**then**	**true**	**undefine**
unique	**until**	**variant**	**when**	**xor**	

G.4 OPERATORS AND THEIR PRECEDENCE

Level	Symbol
12	• (Dot notation for Unqualified_call expressions) – **old** (in postconditions) **strip**
11	**not** unary + unary – All free unary operators –
10	All free binary operators
9	^ (power)
8	* / // (integer division) \\ (integer remainder)
7	binary + binary –
6	= /= (not equal) < > <= >=
5	**and** **and then**
4	**or** **or else** **xor**
3	**implies**

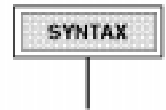
G Reserved words, special symbols, operator precedence

D.12 SPECIFYING EXTERNAL ELEMENTS



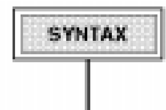
Externals	≜	**external**
		{Language_contribution ";" ...}
Language_contribution	≜	Language ":" File_list
Language	≜	*Eiffel* | *Ada* | *Pascal* |
		Fortran | *C* | *Object* | *Make* |
		Name

D.13 GENERATION

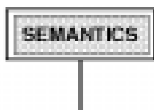


Generation	≜	**generate** {Language_generation ";" ...}
Language_generation	≜	Language [Generate_option] ":" Target
Generate_option	≜	"(" Generate_option_value ")"
Generate_option_value	≜	**yes** | **no**
Target	≜	Directory | File

D.14 VISIBLE FEATURES



Visible	≜	**visible** {Class_visibility ";" ...}
Class_visibility	≜	Class_name [Visibility_adaptation]
Visibility_adaptation	≜	[External_class_rename]
		[Creation_restriction]
		[Export_restriction]
		[External_feature_rename]
		end
External_class_rename	≜	**as** Name
External_feature_rename	≜	**as** Name
Creation_restriction	≜	**creation** {Feature_name "," ...}
Export_restriction	≜	**export** {Feature_name "," ...}
External_feature_rename	≜	**rename** External_rename_list
External_rename_list	≜	{External_rename_pair "," ...}
External_rename_pair	≜	Feature_name **as** Name



Option	Governs	Possible values	Default	Default
assertion	Level of assertion monitoring and execution of Check instructions.	**no**, **require**, **ensure**, **invariant**, **loop**, **check**, **all**	**all**	Monitoring at each level in this list also applies to the subsequent levels (**ensure** implies precondition checking etc.). Value **invariant** means class invariant; **loop** means monitoring of loop invariants and of loop variant decrease; **check** adds execution of **check** instructions; **all** means the same as **check**.
collect	Garbage collection	**no, yes**		
debug	Execution of Debug instructions	**no, yes, all** or a Name representing a Debug_key. Value **yes** means the same as **all**		
optimize	Optimization of generated code	**no, yes, all**, or a Name representing a specific optimization level offered by the compiler.		In the Defaults or Options clause for a given cluster, **yes** governs class-level optimization and **all** means the same as **yes**. In the Ace-level Defaults clause, **yes** governs systemwide optimization, and **all** means the same as **yes** plus class-level optimization.
trace	Generation of run-time tracing information for every call to, and return from, routines of classes to which the option applies.	**no, yes** or **all**. Value **yes** means the same as **all**.		

D.11 SPECIFYING OPTIONS

Defaults	$\triangleq$	**default** {Option_clause ";" ...}
Options	$\triangleq$	**option** {Option_clause ";" ...}
Option_clause	$\triangleq$	Option_tag [Option_mark] [Target_list]
Target_list	$\triangleq$	":" {Class_name "," ...}$^+$
Option_tag	$\triangleq$	Class_tag | System_tag
System_tag	$\triangleq$	*collect* | Free_tag
Class_tag	$\triangleq$	*assertion* | *debug* | *optimize* | *trace* |
		Free_tag
Free_tag	$\triangleq$	Name
Option_mark	$\triangleq$	"(" Option_value ")"
Option_value	$\triangleq$	Standard_value | Class_value
Standard_value	$\triangleq$	**yes** | **no** | **all** | Free_value
Class_value	$\triangleq$	**require** | **ensure** |
		invariant | **loop** | **check** |
		Free_value
Free_value	$\triangleq$	File_name | Directory_name | Name

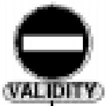
A Target_list may only appear in an Options paragraph, not in a Defaults paragraph. A System_tag may only appear in an Ace-level Defaults clause.

VDOC

		[Defaults]
		[Options]
		[Visible]
		end ["--" **cluster** Cluster_name]

D.8 STORING PROPERTIES WITH A CLUSTER

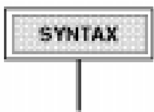
Use	$\triangleq$	**use** File
File	$\triangleq$	Name



VDUC

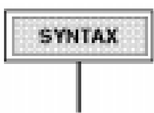
To keep things simple, the Cluster_properties part contained in a Use file may not itself contain a Use paragraph.

D.9 EXCLUDING AND INCLUDING SOURCE FILES



Include	$\triangleq$	**include** File_list
Exclude	$\triangleq$	**exclude** File_list
File_list	$\triangleq$	{File ";" ...}

D.10 ADAPTING CLASS NAMES



Name_adaptation	$\triangleq$	**adapt** Cluster_adaptation_list
Cluster_adaptation_list	$\triangleq$	{Cluster_adaptation ";" ...}
Cluster_adaptation	$\triangleq$	Cluster_ignore |
		Cluster_rename_clause
Cluster_ignore	$\triangleq$	Cluster_name ":" **ignore**
Cluster_rename_clause	$\triangleq$	Cluster_name ":"
		rename Class_rename_list
Class_rename_list	$\triangleq$	{Class_rename_pair "," ...}
Class_rename_pair	$\triangleq$	Class_name **as** Class_name

D.6 ACE STRUCTURE

SYNTAX

Ace	≜	System
		Root
		[Defaults]
		[Clusters]
		[Externals]
		[Generation]
		end ["--" **system** System_name]
System	≜	**system** System_name
System_name	≜	Name
Root	≜	**root**
		Class_name
		[Cluster_mark]
		[Creation_procedure]
Class_name	≜	Name
Cluster_mark	≜	"(" Cluster_name ")"
Cluster_name	≜	Name
Creation_procedure	≜	":" Name

D.7 BASICS OF CLUSTER CLAUSES

SYNTAX

Clusters	≜	**cluster** {Cluster_clause ";" ...}
Cluster_clause	≜	[Cluster_tag]
		Directory_name
		[Cluster_properties]
Cluster_tag	≜	Cluster_name ":"
Directory_name	≜	Name
Cluster_properties		[Use]
		[Include]
		[Exclude]
		[Name_adaptation]

D Specifying systems in Lace

D.1 OVERVIEW

Language processing tools need a specification of where to find the classes and what to do with them.

Such a specification is called an Assembly of Classes in Eiffel, or Ace for short. This appendix presents a notation, the Language for Assembling Classes in Eiffel, or Lace, for writing Aces.

This appendix describes Lace.

D.5 Basic conventions

Here is the list of Lace keywords.

adapt	**all**	**as**	**check**	**cluster**	**creation**
default	**end**	**ensure**	**exclude**	**export**	**external**
generate	**ignore**	**include**	**invariant**	**keep**	**loop**
no	**option**	**require**	**rename**	**root**	**system**
use	**visible**	**yes.**			

Name	$\triangleq$	Identifier | Manifest_string

VDCN

A consistency condition applies to names used in an Ace: the Cluster_name must be different for each cluster. It is valid, however, to use the same identifier in two or more of the roles of Cluster_name, System_name, Class_name.

32 Basic classes

The basic class *BOOLEAN*, *CHARACTER*, *DOUBLE*, *INTEGER* and *REAL* describe arithmetic objects; class *POINTER* describes opaque objects representing addresses to be passed to non-Eiffel software. The specification of these classes may be found in the Proposed Eiffel Library Kernel Standard.

31 Input and output

Classes *FILE* and *STANDARD_FILES* provide input and output facilities. Their specification may be found in the Proposed Eiffel Library Kernel Standard.

30 Persistence and environments

Class *STORABLE* provides a mechanism for storing and retrieving object structures. Its specification may be found in the Proposed Eiffel Library Kernel Standard.

29 Exception facilities

The Kernel Library class *EXCEPTIONS* provides a number of features for fine control of the exception handling mechanism. Their specification may be found in the Proposed Eiffel Library Kernel Standard.

28 Arrays and strings

Arrays and strings are instances of the Kernel Library classes *ARRAY* and *STRING*. The specification of these classes may be found in the Proposed Eiffel Library Kernel Standard.

27 Universal features

The universal features are those of class *GENERAL*. They are specified in the Proposed Eiffel Library Kernel Standard.

C3 • The sequence *%/code/*, where *code* is an unsigned integer, representing the character of code *code*. For example in ASCII *%/59/* represents the character of code 59, which is the semicolon.

The table of special character codes for form C2 is the following._

Special character	*Character Code*	Mnemonic name
@	*%A*	At-sign
BS	*%B*	Backspace
^	*%C*	Circumflex
$	*%D*	Dollar
FF	*%F*	Form feed
\	*%H*	backslasH
tilda	*%L*	tiLda
NL (LF)	*%N*	Newline
‘	*%Q*	[back] Quote
CR	*%R*	[carriage] Return
#	*%S*	Sharp
HT	*%T*	[horizontal] Tab
NUL	*%U*	nUll character
|	*%V*	Vertical bar
%	*%%*	percent
’	*%’*	single quote
"	*%"*	double quote
[	*%(*	opening bracket
]	*%)*	closing bracket
{	*%<*	opening brace
}	*%>*	closing brace

An Integer is a sequence of characters, each of which must be either:

- A decimal digit (0 to 9).
- An underscore (_), which may not be the first character.

If any underscore is present, then there must be three consecutive digits to the right of every underscore, and there must not be any consecutive group of four digits.

SEMANTICS

Underscores, if any, have no effect on the integer value associated with the integer.

25.13 REAL NUMBERS

SYNTAX

A real number is made of the following elements:

- An optional ~~Integer written backwards,~~ giving the integral part. (If this is absent, the integral part is 0.)
- A required "**.**" (dot).
- An optional ~~Integer,~~ which gives the fractional part. (If this is absent, the fractional part is 0.)
- An optional exponent, which is the letter *e* or *E* followed by an optional sign (+ or –) and an Integer. The Integer is required if the *e* or *E* is present. This indicates that the value appearing before the *e* or *E* must be scaled by 10 sup n, where *n* is the given integer.

No intervening character (blank or otherwise) is permitted between these elements. The integral and fractional parts may not both be absent. If underscores are used in either the integral part or the fractional part, they must also appear in the other part, unless it has three digits or less.

SEMANTICS

Underscores, if any, have no effect on the integer value associated with the integer.

25.14 BIT SEQUENCES

SYNTAX

A Bit_sequence is a sequence of digits 0 or 1, followed by a *b* or *B*, with no other intervening characters.

25.15 CHARACTERS

Definition: character

A character is one of the following.

C1 • Any key associated with a printable character, except for the percent key *%*. (The % key plays a special role for cases 2 and 3.)

C2 • The sequence %*k*, where *k* is a one-key code taken from the table given below. This is used to represent special characters such as the Backspace, represented as *%B*, or characters which are not available on all keyboards, or have different codes on different keyboards. An example is the opening bracket: when supported by the keyboard, this character may be entered using form C1 as [; it may also, in all cases, be represented as *%(*.

The printable characters include letters, digits, underscore, the four characters permitted as first character of a Free_operator and other special characters such as *star*, \ and *$*. They exclude Blank, New Line, Backspace and other characters with no external representation.

When used in expressions, the standard operators have various precedence levels, as given in the discussion of expressions; free operators all have the same precedence, higher than that of the standard operators.

Examples of free operators, used as function names, were given in the discussion of features. A simple one is @, used in infix form as a synonym for *item* for array access: *a* @ *i* is the *i*-th element of *i*.

The Basic Libraries contain only a small number of uses of free operators: @ for element access in arrays and strings, # for rotation in bit sequences. Free operators are mostly intended for developers in application areas that have a tradition of specialized notations, such as physics and mathematics. The form of a free operator should attempt to suggest its meaning, just as with a well chosen identifier.

The first character of a Free_operator must be one of only four possibilities, not used by any other construct of the language. As a result, any free operator will stand out clearly from its context, and no confusion or ambiguity is possible.

25.11 STRINGS

Definition: string

A String — a specimen of construct String — is an arbitrary sequence of characters. A Simple_string — a specimen of Simple_string — is a String which consists of at most one line (that is to say, has no embedded new-line character).

Do not confuse String or Simple_string with Manifest_string, seen in the discussion of expressions. A specimen of Manifest_string, a non-terminal construct, is a Simple_ string enclosed in double quotes, as in *"SOME* ~~*STRING". .1W*~~ In the definition of String, a “character” is a legal Eiffel character as defined later in this chapter. This includes in particular:

- Any keyboard key other than *%*.
- Any special character described as *%l* for some appropriate letter *l* (for example *%B* representing the Backspace character).
- A character given by its numerical code under the form %*/code/* (for example *%/35/* for the sharp sign #, which is the character of ASCII code 35).

25.12 INTEGERS

Definition: integer

Integer, a variable lexical construct, describes unsigned integer constants in decimal notation.

Except for underscores, no intervening characters (such as blanks) are permitted between digits.

function, may be used in lieu of a local entity, for example as target of an assignment, and the basic type *INTEGER* may appear at a position where a type is expected.

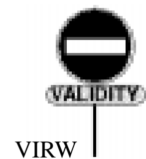
No intervening blanks or other characters are permitted between the letters of a reserved word. Letter case is not significant for reserved words (so that **CLASS**, *result* or even *rEsULt* are permissible forms of some of the above examples.)

25.9 IDENTIFIERS

An Identifier is a sequence of one or more characters, of which the first is a letter and each of the subsequent ones, if any, is a letter, a decimal digit (*0* to *9*) or the underscore character "_"

The definition indicates that the first character of an identifier must be a letter; in particular, an identifier may not begin with an underscore. Also, no intervening blank is permitted within an identifier. The validity constraint is obvious:

An identifier is valid if and only if it is not one of the language's reserved words.



There is no limit to the length of identifiers, and all characters are significant; in other words, to determine whether two identifiers are the same or not, you must take all their characters into account.

Letter case is not significant for letters: if you write two identifiers as *a* and *A*, or *lInKeD_liST* and *LINKED_LIST*, they are considered the same. The recommended style includes some standard conventions: class names and other type names in upper-case (as in *LINKED_LIST*); names of routines, variable attributes and local entities in lower-case (as in *item*); names of constant attributes and predefined entities with an initial upper-case letter and the rest in lower case (as in *Avogadro* or ~~*Result*). Definition: upper name, lower name the form of an identifier written all in upper-case is called the upper name of the identifier; the form all in lower-case is called its lower name.~~

25.10 OPERATORS

When it comes to defining a function with one or two arguments, you may wish to use a prefix or infix name.

Definition: operator

The names that may come in double quotes after infix (or prefix), and will be used in the corresponding operator expressions, are called operators.

There are two kinds of operators: standard and free.

A Free_operator is not as free as the name would seem to suggest:

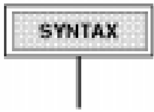
Definition: free operator

A Free_operator is sequence of one or more characters, whose first character is any one of

@ # | &

and whose subsequent characters, if any, may be any printable characters.

In both cases the form is the same: a comment is made of one or more line segments, each beginning with two consecutive dash characters -- and extending to the end of the line.



Comment	$\triangleq$	"--" {Simple_string Comment_break ...}
Comment_break	$\triangleq$	New_line [Blanks_or_tabs] "--"

25.5 TEXT LAYOUT

An Eiffel text is a sequence; each of the elements of the sequence is a break, a comment or a token.

You may always insert a break between two sequence elements without affecting the semantics of the text.

A break is not required between two adjacent elements if one is a comment and the other a token or another comment. Between two successive tokens, a break may be required or not depending on the nature of the tokens.

We may divide tokens into two categories:

Definition: symbol, word

- A **symbol** is either a special symbol of the language, such as the semicolon ";" and the • of dot notation, or a standard (non-free) operator such as + and *star*, but not including the alphabetic operators (such as **or else** and **not**).
- A **word** is any token which is not a symbol.

It is permitted to write two adjacent tokens without an intervening break if and only if one is a word and the other is a symbol.

25.6 TOKEN CATEGORIES

There are two categories of tokens, fixed and variable:

- Fixed tokens have a single, frozen form. They include reserved words such as **class** or *Current*, containing letters only, and special symbols such as := or {, containing non-alphabetic characters.
- Variable tokens are specimens of terminal constructs such as Integer, Identifier or Free_operator.

25.7 RESERVED WORDS

Reserved words include **keywords** and **predefined names**:

- Keywords, such as **class** or **feature**, serve to introduce and delimit the various components of constructs.
- Predefined names come at positions where variable tokens would also be permissible: for example, the predefined entity *Result*, denoting the result of a

25 Lexical components

25.2 CHARACTER CATEGORIES

The discussion will rely on a classification of characters into letters, digits and other categories. To start with, it is useful to have a precise although obvious definition:

Definition: letter, digit, alphabetic, Printable

A letter is one of the twenty-six elements of the Roman alphabet, lower-case or upper-case:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A decimal digit is one of the ten characters *0 1 2 3 4 5 6 7 8 9*

An alphabetic character is a letter, a digit, or an underscore _.

A printable character is any one of the characters listed as printable in the definition of the ASCII character set.

25.3 BREAKS

Definition: break character, break

A break is made of a sequence of one or more of the following characters, known as break characters:

- Blank (also known as space).
- Tab.
- New Line (also known as Line Feed).

25.4 COMMENTS

Definition: comment

Comments may be expected or free:

- An expected comment is a specimen of the construct Comment, appearing in the syntax as an optional component of some construct. An example is the Header_comment of a Routine.
- Free comments, on the other hand, may appear at almost any position in a class text, and are not covered by the syntax productions.

Here *position* is an integer index which must be in the range of the bit sequence (counted from 1); otherwise the first two calls have undefined results, and *eif_bit_ith* returns *EIF_NO_BIT*. Routines *eif_bit_set* and *eif_bit_clear* set the element of index *position* to 1 and 0, respectively. Function *eif_bit_ith* returns an integer, the value of the element of index *position*. Function *eif_bit_length* returns an integer, the length of the sequence. Function *eif_bit_clone* returns an *EIF_BIT*, a fresh copy of the bit sequence passed as argument.

The next Cecil facility enables the C side to access fields of complex objects, corresponding to attributes of the generating classes. To obtain a field of an object, use the macro *eif_field*. You may use the result of *eif_field* in two different ways: as an expression, or “r-value” in C terminology; or as a writable entity, or “l-value”, which may then be the target of an assignment. Such an assignment will re-attach the corresponding object field.

Function *eif_field* takes three arguments. The first is a value of type *EIF_OBJ*, representing an object; do not forget to protect it by *eif_access*. The second is a string giving the name of the desired attribute. The third is one of the following values, describing the type of the attribute:

EIF_BOOLEAN
EIF_CHARACTER
EIF_INTEGER
EIF_REAL
EIF_DOUBLE
EIF_POINTER
EIF_REFERENCE

The result of *eif_field* is undefined if the object does not have a field with the given name and type.

To call an Eiffel function returning a Bit_type result, use this scheme:

EIF_BIT your_bit; EIF_FN_BIT your_function;
...
your_function = eif_fn_bit (function_name, type_id);
your_bit = (your_function) (eif_access (object), actual_1, actual_2, ...)

The last instruction assigns to *your_bit* a reference to the bit sequence returned by the function. *EIF_BIT* describes a pointer type.

Similarly, you may access and modify Bit_type fields as follows:

EIF_BIT your_bit1, your_bit2;
...
your_bit1 = eif_bit_field (eif_access (your_object), "some_bit_attribute");
eif_bit_set_field (eif_access (your_object), "some_bit_attribute", your_bit2);

Here two primitives are needed. Function *eif_bit_field* takes two arguments, an object pointer (returned by *eif_access*) and an attribute name. In contrast with *eif_field*, the result of *eif_bit_field* may only be used as an expression (r-value), not as a writable variable. To change the value of a Bit_type field, use *eif_bit_set_field*, whose last argument is the new value, of type *EIF_BIT*. If the object has no appropriate Bit_type field, *eif_bit_field* returns the value *EIF_NO_BFIELD* and the effect of *eif_bit_set_field* is undefined.

The following primitives are applicable to *your_bit* of type *EIF_BIT*:

~~*eif_bit_set (your_bit, position) eif_bit_clear (your_bit, position) eif_bit_ith (your_bit, position) eif_bit_length (your_bit) eif_bit_clone (your_bit)*~~

it in a C variable *your_id*, use the function *eif_type_id*, which returns a result of type *EIF_TYPE_ID*:

EIF_TYPE_ID your_id;

...

your_id = *eif_type_id* (*"CLASSNAME"*);

If the class is generic, replace the last instruction by

your_id = *eif_generic_id* (*"CLASSNAME", gen1, gen2, ...*);

where *gen1, gen2, ...* are type-ids corresponding to the desired actual generic parameters. Function *eif_generic_id* has a variable number of arguments; the number of arguments following the first one ("*CLASSNAME*") must match the number of formal generic parameters of the class of name *CLASSNAME*.

The result returned by *eif_type_id* or *eif_generic_id* describes a type which is expanded if and only if *CLASSNAME* is declared in Eiffel as **expanded class**. To force a result describing an expanded type, apply *eif_expanded* to the result of either function; the result is another type-id. All these functions return as result the error code *EIF_NO_TYPE* if they cannot compute a type-id (no class with the given name in the universe, more than one class, wrong number of generic parameters). It is possible to create an object from C, using the function *eif_create* which takes an *EIF_TYPE_ID* argument and returns an *EIF_OBJ*. Here is the list of functions used for calling the various kinds of Eiffel features from C, with their types and template arguments:

EIF_PROC eif_proc (*routine_name, type_id*)

EIF_FN_BOOL eif_fn_bool (*routine_name, type_id*)

EIF_FN_CHAR eif_fn_char (*routine_name, type_id*)

EIF_FN_INT eif_fn_int (*routine_name, type_id*)

EIF_FN_REAL eif_fn_real (*routine_name, type_id*)

EIF_FN_DOUBLE eif_fn_double (*routine_name, type_id*)

EIF_FN_POINTER eif_fn_pointer (*routine_name, type_id*)

EIF_FN_REF eif_fn_ref (*routine_name, type_id*)

In all cases the arguments are a string, representing a routine name, and a type-id (obtained through *eif_type_id* or *eif_generic_ic*):

char starroutine_name; *EIF_TYPE_ID type_id*;

SEMANTICS

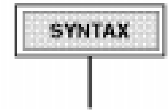
These functions look for a routine of name *routine_name* in the base class of the type corresponding to *type_id*. If no such routine exists, the result is a null pointer. Otherwise it is a pointer to a C function representing the desired routine; you may then call that function on appropriate arguments.

SEMANTICS

There is a major difference between the Eiffel call and its C emulation: Cecil **does not apply dynamic binding**. What you get from *eif_proc* or one of its sisters is a pointer to a function representing the exact Eiffel routine of the given name in the given class. In the presence of polymorphism and redeclaration, the Eiffel call may trigger a different version of *go* depending on the type of the object attached to *your_list* at the time the call is executed. The C form will always call the same version, regardless of the object's type.

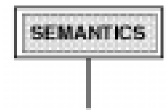
24 Interfaces with other languages

24.3 EXTERNAL ROUTINES



External	$\triangleq$	**external** Language_name [External_name]
Language_name	$\triangleq$	Manifest_string
External_name	$\triangleq$	**alias** Manifest_string

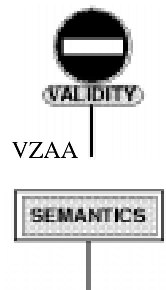
24.5 ARGUMENT AND RESULT TRANSMISSION



For external routines, follow the ~~the~~ semantics of direct reattachment, interpreted as if each formal argument were declared with **exactly** the same type as the corresponding actual.

24.6 PASSING THE ADDRESS OF AN EIFFEL FEATURE

An argument of the Address form is of type *POINTER*.



The validity constraint on actual arguments of the Address form is clause 4 of the argument validity rule, which makes *$ f* valid as actual argument to a call if and only if *f* is the final name of a feature of the enclosing class, and that feature is not a constant attribute (which has no address).

If the rule is satisfied, the feature will have a version *df* applicable to the current object: this is the version of *f* for the current object's generator (taking into account possible renaming and redefinition). The value passed for attachment to the corresponding formal argument is the address of *df*. This applies to both routines and variable attributes; for an attribute, the call will pass the address of the field corresponding to *df* in the current object.

24.7 THE CECIL LIBRARY

This section describes a library for interfacing with C software: Cecil.

The Cecil library contains macros, functions, types and error codes. All have names beginning with either *eif_* (functions and macros) or *EIF_* (types and error codes); examples are the function *eif_type_id* and the type *EIF_PROC*, explained below. Their declarations appear in a C "header file", *cecil•h*, which you may add to a C program through the C preprocessor directive

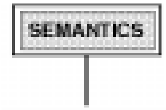
#include <cecil•h>

Let us now look at the principal facilities declared in *cecil•h*. First of all, the C side will need to refer to Eiffel types. It will know a type through a "type-id", of type *EIF_TYPE_ID*. To obtain a type-id for a non-generic class of name *CLASSNAME* and record

to attributes a_1, ..., a_n, appearing in an order which only depends on class *C* (that is to say, is the same for all possible values of the object CO).

Bit_constant	≜	Bit_sequence

23.17 CHARACTER CONSTANTS



The value of a Character_constant is its middle Character.

23.18 MANIFEST STRINGS

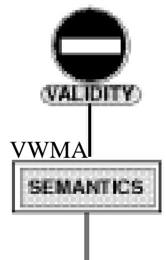
A Manifest_string is valid if and only if it satisfies the following two conditions:

1 • None of the characters of its associated Simple_string is a double quote.

2 • In the extended form, no characters other than blanks or tabs may appear before the initial % sign on the second and subsequent lines.

23.20 MANIFEST ARRAYS

Manifest_array	≜	"<<" Expression_list ">>"
Expression_list	≜	{Expression "," ...}



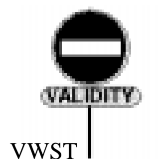
Manifest Array rule

A Manifest_array <<e_1, e_2, ... e_n>> is a valid expression of type *ARRAY* [*T*] if and only if the type of every e_i conforms to *T*.

The value of a Manifest_array made of *N* expressions is an array of bounds 1 and *N*, whose elements are the values of the successive expressions in the Manifest_array. In this definition, an “array” is an instance of the Kernel Library class *ARRAY*.

23.21 STRIP EXPRESSIONS

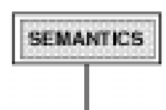
Strip	≜	**strip** "(" Attribute_list ")"
Attribute_list	≜	{Identifier "," ...}



A Strip expression appearing in a class *C* is valid if and only if its Attribute_list satisfies the following two conditions:

1 • Every Identifier appearing in the list is the final name of an attribute of *C*.

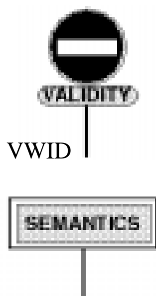
2 • No Identifier appears twice in the list.



Consider the evaluation of a Strip expression as part of a call to a routine *r*, whose origin is a class *C*. Let a_1, ..., a_n be the set containing all the attributes of *C* except for those listed in the Attribute_list, if present, of the Strip expression. Let CO be the current object for the call (CO is an instance, not necessarily direct, of a type based on *C*). Then the value of the expression is an array whose elements are the fields of CO corresponding

An Equality expression involving two bit sequences yields true if and only if, after right-padding if necessary, they are bit-by-bit identical.

23.11 ENTITIES



VWID

Identifier rule

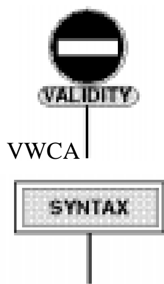
An Identifier appearing in an expression as part of the text of a routine *r* in a class *C*, either by itself or as the target or actual argument of a Call, must be the name of a feature of *C*, a local entity of *r*, or a formal argument of *r*.

The value of an entity of each possible form, evaluated during a call to the enclosing routine, is defined as follows:

- The value of a Local entity (including *Result*) results from the successive instructions that may have been applied to the entity since the default initializations, performed anew on each call.
- The value of a routine's Formal argument is obtained, according to the rules of direct reattachment, from the value of the corresponding actual argument at the time of the current call. This value may not change for the duration of that call (although fields of the attached object, if any, may change).
- The value of *Current* is the current object.

23.12 CONSTANTS

Constant	=	Manifest_constant | Constant_attribute
Constant_attribute	=	Entity



VWCA

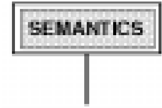
A Constant_attribute appearing in a class *C* is valid if and only if its Entity is the final name of a constant attribute of *C*.

Manifest_constant	$\triangleq$	Boolean_constant | Character_constant | Integer_constant | *Real_constant* | Manifest_string | Bit_constant
Sign	$\triangleq$	"+" | "–"
Integer_constant	$\triangleq$	[*Sign*] Integer
Character_constant	$\triangleq$	"'" Character "'"
Boolean_constant	$\triangleq$	**true** | **false**
Real_constant	$\triangleq$	[*Sign*] Real
Manifest_string	$\triangleq$	'"' Simple_string '"'

Definition: multiary operator

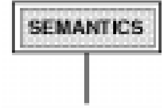
> Some binary operators are actually "multiary" — that is to say, may take three or more operands, whose types all conform to the type of the first — while other operators are limited to two arguments.

The semantics of an Operator_expression is simply the semantics of calls: the value of an operator expression is the value that would be returned by the equivalent dot form. This is complemented by a special rule seen below, semi-strict evaluation, for boolean operators.



23.7 ORDINARY BOOLEAN OPERATORS

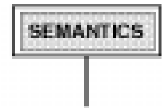
The value of **not** *a* is true if and only if *a* has value false. The others are binary operators; the value they yield when applied to a first operand of value *v1* and a second operand of value *v2* is defined as follows:



- For **and**: true if and only if both *v1* and *v2* are false.
- For **or**: false if and only if either *v1* or *v2* is false.
- For **xor**: true if and only if *v1* and *v2* have different values. In other words, *a* **xor** *b* has the same value as (*a* **or** *b*) **and not** (*a* **and** *b*).

23.8 SEMI-STRICT BOOLEAN OPERATORS

For operands of values *v1* and *v2* these operators yield the following results:

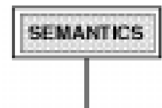


- **and then**: false if *v1* is false, otherwise the value of *v2*.
- **or else**: true if *v1* is true, otherwise the value of *v2*.
- **implies**: true if *v1* is false, otherwise the value of *v2*. (In other words, *a* **implies** *b* has the same value as **not** *a* **or else** *b*.)

23.9 ARITHMETIC OPERATORS AND THE BALANCING RULE

Arithmetic Expression Balancing rule

In determining the equivalent dot form of a Binary expression involving operands of arithmetic types (one or more of *DOUBLE*, *REAL* and *INTEGER*), first convert all operands to the heaviest operand type occurring in the expression.



23.10 OPERATIONS ON BIT SEQUENCES

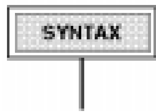
If *b* is a bit sequence, **not** *b* is a bit sequence of the same length, with a one at every position where *b* has a zero and conversely.

If *sa* and *sb* are bit sequences, each of the binary operators pads the shorter operand with zeros on the right if necessary to reach the number of bits, N, of the larger. For any position *i* ($1 \leq i \leq N$) let ~~*a* , *b* ,~~ *c* be the *i*-th bits of ~~*sa* ,~~ *sb*, *sc*. Then *c* is a one if and only if:

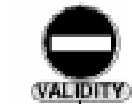
- For **and**: *a* and *b* are ones.
- For **or**: *a* or *b* or both are ones.
- For **implies**: *a* is a zero, or *b* is a one, or both of these conditions.
- For **xor**: one among *a* and *b* is a one and the other is a zero.

23 Expressions and constants

23.2 GENERAL FORM OF EXPRESSIONS



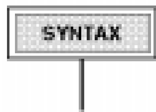
Expression	$\triangleq$	Call | Operator_expression | Equality |
		Manifest_constant | Manifest_array |
		Old | Strip
Boolean_expression	$\triangleq$	Expression



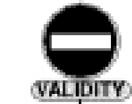
VWBE

A Boolean_expression is valid if and only if it is an Expression of type *BOOLEAN*.

23.3 EQUALITY EXPRESSIONS

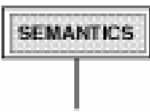


Equality	$\triangleq$	Expression Comparison Expression
Comparison	$\triangleq$	"=" | "/="



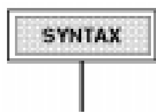
VWEQ

An Equality expression is valid if and only if either of its operands conforms to the other.

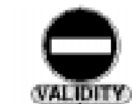


The expression e /= f has value true if and only if $e = f$ has value false.

23.5 OPERATOR EXPRESSION SYNTAX AND PRECEDENCE RULES



Operator_expression	$\triangleq$	Parenthesized | Unary_expression |
		Binary_expression
Parenthesized	$\triangleq$	"(" Expression ")"
Unary_expression	$\triangleq$	Prefix_operator Expression
Binary_expression	$\triangleq$	Expression Infix_operator Expression



VWOE

23.6 VALIDITY AND SEMANTICS OF OPERATOR EXPRESSIONS

An Operator_expression is valid if and only if its equivalent dot form is a valid Call.

4 • If a routine of *C* contains an Assignment of target *x* and source *e*, the dynamic type set of *x* for *T* includes (recursively) every member of the dynamic type set of *e* for *T*.

5 • If a routine of *C* contains an Assignment_attempt of target *x*, with type *U*, and source *e*, the dynamic type set of *x* for *T* includes (recursively) every type conforming to U_T which is also a member of the dynamic type set of *e* for *T*.

6 • If a routine of *C* contains a call *h* of target *ta*, *U* is (recursively) a member of the dynamic type set of *ta* for *T*, and *tf* is the version of the call's feature in the base class of *U*, then the dynamic type set for *U* of any formal argument of *tf* includes every member of the dynamic type set for U_T of the corresponding actual argument in *h*.

7 • If *h*, *tf* and *U* are as in case 6 and *tf* is an attribute or function, the dynamic type set of *h* for *T* includes (recursively) every member of the dynamic type set for U_T of the *Result* entity in *tf*.

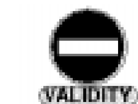
~~**Array type rule**~~

~~To study the effect of array manipulations on dynamic type sets, assume that in class *ARRAY* feature *item* is an attribute, and that *put* (*v*, *i*) and *force* (*v*, *i*) are both implemented as~~

~~*item* := *v*~~

22 Type checking

22.9 THE CALL VALIDITY RULE

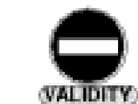


VUGV

Call rule

A call is valid if and only if it is both class-valid and system-valid.

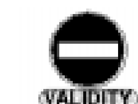
Consider a single-dot call with target *x*, appearing in a class *C*. Let *S* be the type of *x*. Then:



VUCS

1 • The call is **class-valid** if it is export-valid and argument-valid for *S*.

2 • The call is **system-valid** if for any element *D* of the dynamic class set of *x* it is export-valid and argument-valid for *D*.

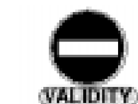


VUEX

A call appearing in a class *C*, having *fname* as the feature of the call, is export-valid for a class *D* if and only if it satisfies either of the following two conditions.

1 • The call is an Unqualified_call and *fname* is the final name of a feature of *C*.

2 • The call has at least one dot, *D* has a feature of name *fname* which is available to *C*, and the call's target is either a valid entity of *C* or (recursively) a call which is export-valid for *D*.



VUAR

Consider an export-valid call of target *target* and feature name *fname* appearing in a class *C*. (For an Unqualified_call take *target* to be *Current*.) Let *ST* be the type of *target*, *S* the base class of *ST*, and *sf* the feature of final name *fname* in *S*. Let *D* be a descendant of *S*, and *df* the version of *sf* in *D*. The call is argument-valid for *D* if and only if it satisfies the following four conditions:

1 • The number of actual arguments is the same as the number of formal arguments declared for *df*.

2 • Every actual argument, if any, conforms to the ~~the~~ corresponding formal argument of *df*.

3 • If *target* is itself a Call, it is (recursively) argument-valid for *D*.

4 • If any of the actual arguments is of the Address form *$ fn*, *fn* is the final name of a feature of *C* which is not a constant attribute.

The dynamic type sets of the expressions, entities and functions of a system result from performing all possible applications of the following rules to every Class_type *T*, of base class *C*, used in the system.

1 • If a routine of *C* contains a creation instruction, with target *x* and creation type *U*, the dynamic type set of *x* for *T* is $\{U_T\}$.

2 • The dynamic type set for *T* of an occurrence of *Current* in the text of a routine of *C* is $\{T\}$.

3 • For any entity or expression *e* of expanded type appearing in the text of *C*, if the type *ET* of *e* is expanded, the dynamic type set of *e* for *T* is $\{ET_T\}$. (Rules 4 to 7, when used to determine elements of the dynamic type set of some *e*, assume that *e*'s type is not expanded.)

effect of a qualified call of the form *s*•*u*, where *s* is an expression and *u* is an Unqualified_call, is defined (recursively) as the effect of a call of the form *v*•*u*, executed after the assignment *v* := *target*•*s*, *v* being a Writable entity used only for this definition.

D5 • If *df* is a function, the call is syntactically an expression; the value of that expression is the value of the entity *Result* on termination of the function's execution.

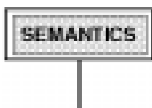
D6 • If the values of any local entities have been saved under D2, restore these entities to their earlier values. This terminates the execution.

2 • If *cr* executes any construct other than a call, the current object and current routine remain the same.

3 • If *cr* executes a qualified call of the form *target*•*fname* (...) where the value *target_value* of *target* is attached to an object OD, then for the duration of the call OD becomes the new current object and *df*, the routine deduced from *fname* as discussed earlier, becomes the new current routine. When the qualified call terminates, the earlier CO and *cr* resume their roles as current object and current routine.

4 • When *cr* executes an unqualified call, the current object remains the same, and there is a new current routine for the duration of the call as in case 3.

21.14 SEMANTICS OF CALLS



Consider the execution, at a certain run-time instant, of a call

target•*fname* (y_1, ..., y_n)

To define its effect, call *target_value* the value of *target* at that instant.

The first possibility is for *target_value* to be void. Then the call cannot be executed correctly; it will fail, triggering an exception.

For the exception raised in this case, the Kernel Library class *EXCEPTIONS* introduces the integer code *Void_call_target*.

The rest of this section assumes that *target_value* is not void. Then *target_value* is attached to some object OD, which must be a direct instance of some type *DT* (for "dynamic type") based on some class *D*. Let *ST* be the type of expression *target* and *S* the base class of *ST*. The rules of reattachment indicate that *DT* conforms to *ST*, and that *D* is a descendant of *S*. Also, because *DT* has a direct instance OD, *D* must be an effective class.

If the call is valid, the constraint on calls implies that *fname* is the final name of a feature *sf* of class *S*, available to the class which contains the call. Let *df* be the version of *sf* in *D*; dynamic binding means that the effect of the call is determined by *df*, not *sf*. If *df* is an external routine, the effect of the call is to execute that routine on the actual arguments given, if any, according to the rules of the language in which it is written.

There remains to cover the case in which *df* is a non-external non-once routine (with a Routine_body beginning with the keyword **do**). Then the effect of the call is the effect of the following sequence of steps.

D1 • If *df* has arguments, attach every formal argument to the value of the corresponding actual argument, applying the semantics of direct reattachment.

D2 • If *df* has any local entities, save the current values of these entities if any call to *df* has been started but not yet terminated; then initialize each local entity to the default value of its base type.

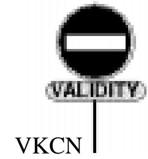
D3 • If *df* is a function, initialize the predefined entity *Result*, again according to the default initialization rules.

D4 • Execute the body of *df*. In this execution, evaluation of the entity *Current* will return *target_value*; the effect of an Unqualified_call *u*, where *u* is neither *Current* nor *Result*, is defined (recursively) as the effect of the qualified call *target*•*u*; and the

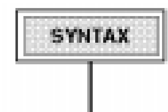
21 Feature call

21.3 USES OF CALLS

- If the feature is an attribute or a function, the Call is syntactically an expression.
- If the feature is a procedure, the Call is an instruction.



21.6 FORM OF A CALL



Call	≜	[Parenthesized_qualifier] Call_chain
Parenthesized_qualifier	≜	~~Parenthesized\0"~~Parenthesized "•"
Call_chain	≜	{Unqualified_call "•" ...}$^+$
Unqualified_call	≜	Entity [Actuals]
Actuals	≜	"(" Actual_list ")"
Actual_list	≜	{Actual "," ...}
Actual	≜	Expression | Address
Address	≜	"$" Address_mark
Address_mark	≜	Feature_name | *Current* | *Result*

21.12 EXECUTING THE BODY

To execute (or “run”) a system on a machine means to get the machine to apply a creation instruction to the system’s root class.

Definition: current object

In all but trivial cases, the root’s creation procedure will create more objects and execute more calls. This extremely simple semantic definition of system execution has as its immediate consequence to yield a precise definition of the current object and current routine. At any time during execution, the current object CO is the object to which the latest non-completed routine call applies, and the current routine *cr* is the feature of that call. They may be defined precisely as follows:

1 • At the start of the execution of a system, CO is the **root object** (the instance of the root class whose creation is the first act of system execution) and *cr* is the selected creation procedure. (If the root class has no creation procedure, execution terminates immediately.)

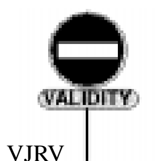
Definition: dynamic type set, dynamic class set

The set of possible dynamic types for an entity or expression *x* is called the dynamic type set of *x*. The set of base classes of these types is called the dynamic class set of *x*.

20.14 RULES ON ASSIGNMENT ATTEMPT

Assignment_attempt	$\triangleq$	Writable "?=" Expression

Assignment Attempt rule



VJRV

An Assignment_attempt is valid if and only if the type of the target entity is a reference type.

The effect of an Assignment_attempt of source *y* and target *x*, of type *TX*, is the following:

1 • If *y* is attached to an object whose type conforms to *TX*, then the effect is that of a direct reattachment of *y* to *x*, as given earlier in this chapter.

2 • If *y* is void or attached to an object whose type does not conform to *TX*, the effect is to make the value of *x* void.

20.17 SEMANTICS OF EQUALITY

There are in fact two operators to consider here: equality = and inequality /=. However only one semantic definition is needed, since the effect of *x* /= *y* is defined in all cases to be that of

not (*x* = *y*)

TYPE OF FIRST→ *TYPE OF SECOND*↓	**Reference**	**Expanded**
Reference	[1] Reference equality	[3] *standard_equal*
Expanded	[2] *standard_equal*	[4] *standard_equal*

20 Reattaching entities

20.4 SYNTAX AND VALIDITY OF DIRECT REATTACHMENT

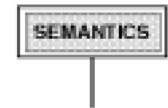
Assignment	$\triangleq$	Writable ":=" Expression

Assignment rule

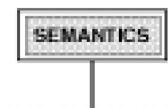
VJAR

An Assignment is valid if and only if its source expression conforms to its target entity.

20.5 SEMANTICS OF DIRECT REATTACHMENT



SOURCE TYPE → *TARGET TYPE* ↓	**Reference**	**Expanded**
Reference	[1] Reference reattachment	[3] Clone
Expanded	[2] Copy (will fail if source is void)	[4] Copy



For the exception raised in case 2 if the value of *y* is void, the Kernel Library class *EXCEPTIONS* introduces the integer code *Void_assigned_to_expanded*.

20.9 POLYMORPHISM

Definition: dynamic type

The dynamic type of an entity or expression *x*, at some instant of execution when *x* is not void, is the type of the object to which *x* is attached. This should not be confused with the "type" of *x* (called its *static type* if there is any ambiguity), which for an entity is the type with which it is declared, and for an expression is the type deduced from the types of its constituents.

Definition: polymorphism

The ability to have more than one dynamic type is called polymorphism; an entity or expression which has two or more possible dynamic types (that is to say, which may become attached at run time to objects of two or more types) is itself a polymorphic entity. Only entities or expressions of reference types may be polymorphic.

3 • .If *TY* does not conform to *TX*, the result is false. Cases 4 to 6 assume that *TY* conforms to *TX*.

4 • If *TX* is *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, *DOUBLE* or *POINTER*, the result is true if and only if OX and OY are the same value, after possible conversion to the heavier type if OX and OY are different arithmetic types.

5 • If OX and OY are special objects (strings or arrays), that is to say, sequences of values, the result is true if and only if the sequences have the same length, and every field in one is (recursively) equal to the field at the same position in the other.

6 • Otherwise OX and OY are standard complex objects, and conformance of *TY* to *TX* implies that for every field of OX there is a corresponding field in OY. Then the result is true if and only if every reference field of OX is attached to the same object as the corresponding field in OY, and every object field of OX, coming from an expanded attribute in *TX*'s base class, is (recursively) equal to the corresponding field in OY.

19.8 DEEP EQUALITY

Two references *x* and *y* are deep-equal if and only if they are either both void or attached to deep-equal objects.

Two objects OX and OY are deep-equal and only if they satisfy the following four conditions:

1 • OX and OY have the same exact type.

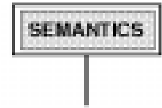
2 • The objects obtained by setting all the reference fields of OX and OY (if any) to void references are equal.

3 • For every void reference field of OX, the corresponding field of OY is void.

4 • For every non-void reference field of OX, attached to an object PX, the corresponding field of OY is attached to an object PY, and it is possible (recursively) to show, under the assumption that OX is deep-equal to OY, that PX is deep-equal to PY.

19 Duplicating and comparing objects

19.2 COPYING AN OBJECT



Assume *copy* has not been redefined and consider a call *x*• *copy* (*y*). As with any call, the target *x* must be non-void; the first precondition clause of *copy* states that *y* must also be non-void. Let OX and OY be the attached objects at the time of the call.

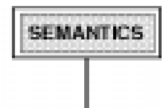
1 • If OX and OY are bit sequences, the conformance rule on Bit_type requires OX to be at least as long as OY. The call then copies onto OX the contents of OY, extended with zeros on the left if OY is shorter than OX. Cases 2 to 4 assume that OX and OY are not bit sequences.

2 • If the types of OX and OY are basic types (*BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, *DOUBLE* or *POINTER*), then OX is identical to OY or is a heavier type. Then the call copies the value of OY onto OX, after conversion to the heavier type if necessary.

3 • If OX and OY are special objects, that is to say sequences of values (strings or arrays), the call copies the value of OY onto OX. The implementation must ensure that whenever this occurs the size of OX is at least as large as the size of OY.

4 • In the remaining cases OX and OY are standard objects, made of zero or more fields, and the second precondition clause implies that the type of OY is a descendant of the type of OX; as a result, for every field of OX there is a corresponding field in OY. Then the call copies onto every field of OX the corresponding field of OY.

19.3 CLONING AN OBJECT

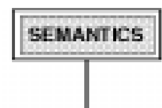


Here is the definition of the semantics of a call *clone* (*y*):

1 • If the value of *y* is void, the call returns a void value.

2 • If the value of *y* is attached to an object OY, the call returns a newly created object of the same type as OY, initialized by applying *copy* to that object with OY as source.

19.7 OBJECT EQUALITY



Here is the precise result that the standard version of *equal* must return when applied to two values *x* and *y*.

1 • If any one of *x* and *y* is void, the result is true if the other is also void too, and false otherwise. Cases 2 to 6 assume that both arguments are attached to respective objects OX and OY of types *TX* and *TY*.

2 • If OX and OY are bit sequences, the result is true if and only if the two sequences, with the shorter one being extended with zeros on the left to match the length of the longer one if necessary, are bit-by-bit identical. Cases 3 to 6 assume that OX and OY are not bit sequences.

- For *BOOLEAN*: the boolean value false.
- For *CHARACTER*: the null character.
- For *INTEGER*, *REAL* or *DOUBLE*: the integer, single precision or double precision zero.
- For *POINTER*: a null pointer.
- For a Bit_type of the form *BIT* N: a sequence of N zeros.

2 • *C* is an effective class.

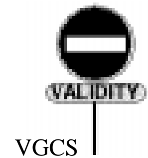
3 • If the Type part is present, the type that it contains (which is *T*) conforms to the type of *x*, and is a reference type.

4 • If *C* does not have a Creators part, there is no Creation_call part.

5 • If *C* has a Creators part, there is a Creation_call part, and the call would be argument-valid if it appeared in the text of *C*.

6 • If case 5 holds and *f* is the feature of the Creation_call, then *f* is available for creation to *X*.

18.10 CREATION VALIDITY (SYSTEM-LEVEL)

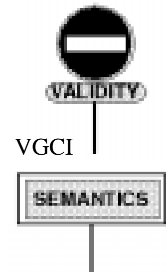


A Creation instruction is **system-valid** if and only if it satisfies one of the following two conditions:

1 • The creation type is explicit (in other words, the instruction begins with ! *T* ! ... for some type *T*).

2 • The creation type is implicit (in other words, the instruction begins with !!...) and every possible dynamic type *T* for *x*, with base class *C*, satisfies conditions 1 to 6 of the Creation Instruction ~~rule (page \n(9g).~~ In applying conditions 5 and 6, the feature of the call, *f*, must be replaced by its version in *C*.

A Creation instruction is valid if and only if it is both class-valid and system-valid.



18.11 CREATION SEMANTICS

With the above validity rules, we can define the precise semantics of a Creation instruction. Consider such an instruction with target *x* and creation type *T*. If *T* is a reference type, the effect of executing the instruction is the following sequence of steps:

1 • If there is not enough memory available to create a new direct instance of *T*, trigger an exception in the routine that executed the instruction. Steps 2 to 5 do not apply in this case.

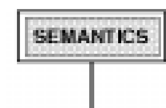
2 • Create a new direct instance of *TC*.

3 • Assign a value to every field of the new instance: for a field corresponding to a constant attribute, the value defined in the class text; for a field corresponding to a variable attribute, the default value of the attribute's type, according to the rules given below.

4 • If the Creation instruction includes a Creation_call, that is to say an Unqualified_ call, execute that call on the resulting object.

5 • Attach *x* to the object.

18.13 DEFAULT INITIALIZATION VALUES



Consider a field of a newly created object, corresponding to an attribute of type *FT* in the base class of the object's type. The default initialization value *init* for the field is determined as follows according to the nature of *FT*.

- For a reference type: a void reference.

18 Creating objects

18.8 CREATION SYNTAX

Creators	$\triangleq$	**creation** {Creation_clause **creation** ...}$^+$
Creation_clause	$\triangleq$	[Clients] [Header_comment]
		Feature_list

Creation_clause rule

VGCP

A Creation_clause appearing in the Creators part of a class *C* is valid if and only if it satisfies the following five conditions, the last four for every Feature_identifier *fname* in the clause's Feature_list:

1 • *C* is effective.

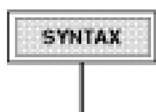
2 • *fname* appears only once in the Feature_list.

3 • *fname* is the final name of a procedure of *C*; let *cp* be that procedure.

4 • *cp* is not a once routine.

5 • If *C* is expanded, *cp* has no arguments, and no other Feature_identifier appears in the Feature_list.

Definition: creation procedure



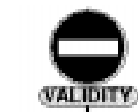
Creation	$\triangleq$	"!" [Type] "!" Writable [Creation_call]
Creation_call	$\triangleq$	"•" Unqualified_call

Definition: creation Type

To discuss the validity and semantics of the Creation instruction it is useful to introduce the notion of creation type of a Creation instruction. The creation type is the optional Type appearing in the instruction (between exclamation marks) if present; otherwise it is the base type of the target.

18.9 CREATION VALIDITY (CLASS-LEVEL)

Creation Instruction rule



VGCC

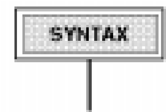
Consider a Creation instruction appearing in a class *X*. Let *x* be the target of the instruction, *T* its creation type, and *C* the base class of *T*.

The instruction is **class-valid** if and only if it satisfies the following conditions:

1 • *T* is not a Formal_generic_name (that is to say, a formal parameter of the class where the instruction appears).

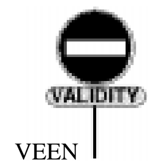
4 • *Current*, the predefined entity used to represent a reference to the current object (the target of the latest not yet completed routine call).

Unique	$\triangleq$	**unique**



Entity rule

An occurrence of an entity *e* in the text of a class *C* (other than as feature of a qualified call) is valid if and only if it satisfies one of the following conditions:



VEEN

1 • *e* is the final name of an attribute of *C*.

2A • The occurrence is in a Local_declarations, Routine_body, Postcondition or Rescue part of a Routine text for a function, and *e* is the Local entity *Result*.

2B • The occurrence is in a Local_declarations, Routine_body or Rescue part of a Routine text for a routine *r*, and the Local_declarations part for *r* contains an Entity_declaration_list including *e* as part of its Identifier_list.

3 • The occurrence is in a Feature_declaration for a routine *r*, and the Formal_arguments part for *r* contains an Entity_declaration_list including *e* as part of its Identifier_list.

4 • *e* is *Current*.

- For *BIT n*, with $n \geq 0$: all the sequences of *n* binary (zero or one) values (none if *n* = 0).

17.7 FIELDS OF COMPLEX OBJECTS

Definition: complex class, complex type, field

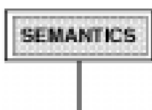
Every class other than *BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, *DOUBLE* and *POINTER* is said to be a complex class. Any type whose base class is complex is itself a complex type, and its instances are complex objects.

Consider a class type *TC*, of base class *C*, and an attribute *a* of class *C*, with *TA* being the type of *a*. The possible values for the field corresponding to attribute *a* in a direct instance of *TC* depend on the nature of *TA*. There are three possible cases for *TA*:

1 • Reference type. (This also covers the case of an anchored type, of the form **like** *x*, which has a class type as base type.)

2 • Expanded type.

3 • Formal generic parameter of class *C*.



In case 1, the field corresponding to attribute *a* is a reference. That reference may be void, or it may be attached to an instance of *TA*'s base type — not necessarily a direct instance.

Definition: sub-object

In case 2, the field corresponding to attribute *a* is an instance of the expanded type *TA*. In other words, the field is itself an object, called a sub-object of the enclosing object. Depending on the precise nature of *TA*, the sub-object may be of various forms:

- *TA* may be a basic type, in which case the sub-object is a basic object of the corresponding type; the figure shows fields of type *INTEGER*, *DOUBLE* and Bit_ type.

Definition: composite object

- If *TA* is a non-basic expanded type, the sub-object is itself a complex object. In this case the enclosing complex object is said to be composite.

17.9 EXPRESSIONS AND ENTITIES

Definition: entity

In a class text, four kinds of entity may appear:

1 • Final names of attributes of the class.

2 • Local entities of routines, including the predefined entity *Result* for functions.

3 • Formal routine arguments.

17 Objects, values and entities

17.2 OBJECTS

Definition: standard object, special object

There are two kinds of object, standard and special:

- A standard object is the direct result of a Creation instruction or clone operation executed by the system.
- A special object is a sequence of values, all compatible with a given type. It may be a **string** or an **array**. In a string, the values are all characters; in an array, they are either all references, or all direct instances of a single type.

17.4 INSTANCES OF A CLASS

Definition: instance of a class

An instance of a class *C* is an instance of any type *T* based on *C*, and similarly for direct instances.

17.5 INSTANCES AND DIRECT INSTANCES OF A TYPE

Definition: instances of a type

The instances of a type *TX* are the direct instances of any type conforming to *TX*.

Definition: ~~diirect~~ instances of non-basic expanded types

Let *TX* be an expanded type which is neither one of the basic types (*BOOLEAN*, *CHARACTER*, *INTEGER*, *REAL*, *DOUBLE*, *POINTER*) nor a Bit_type. The direct instances of *TX* are the the direct instances of the base type of *TX*.

17.6 DIRECT INSTANCES OF BASIC TYPES

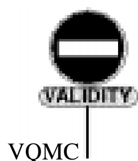
Direct instances of basic types

The direct instances of the basic types are the following.

- For *BOOLEAN*: the boolean values true and false.
- For *CHARACTER*: any character.
- For *INTEGER*: all the integer values which may be represented on *Integer_bits* bits.
- For *REAL*: all floating-point values which may be represented on *Real_bits* bits.
- For *DOUBLE*: all the floating-point values which may be represented on *Double_bits* bits.
- For *POINTER*: all possible feature addresses, for transmission to non-Eiffel routines.

16 Attributes

16.5 CONSTANT ATTRIBUTES WITH MANIFEST VALUES

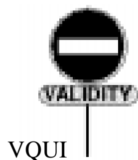


A declaration of a feature *f* introducing a manifest constant is valid if and only if the Manifest_constant *m* used in the declaration matches the type *T* declared for *f* in one of the following ways:

- *m* is a Boolean_constant and *T* is *BOOLEAN*.
- *m* is a Character_constant and *T* is *CHARACTER*.
- *m* is an Integer_constant and *T* is *INTEGER*.
- *m* is a Real_constant and *T* is *REAL* or *DOUBLE*.
- *m* is a Manifest_string and *T* is *STRING*.
- *m* is a Bit_constant consisting of exactly M bits for some positive integer M, and *T* is *BIT* M for the same M.

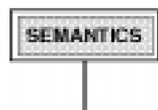
16.6 UNIQUE ATTRIBUTES

Unique $\triangleq$ **unique**



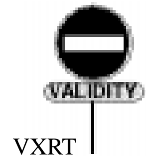
A declaration of a feature *f* introducing a Unique constant is valid if and only if the type *T* declared for *f* is *INTEGER*.

Unique Declaration rule



The value of an attribute declared as unique is a positive integer. If two unique attributes are introduced in the same class, their values are different. Furthermore, unique attributes declared as part of the same Feature_declaration are guaranteed to have consecutive values, in the order given.

A Retry instruction is valid if and only if it appears in a Rescue clause.



15.9 SEMANTICS OF EXCEPTION HANDLING

Definition: rescue block

Any Internal routine *r* of a class *C* has a **rescue block** *rb*, which is a Compound defined as follows:

1 • If *r* has a Rescue clause, *rb* is the Compound contained in that clause.

2 • If *r* has no Rescue clause, *rb* is a Compound made of a single instruction: a call to the version of *default_rescue* in *C*.

An exception triggered during an execution of a routine *r* leads, if it is neither ignored nor continued, to the following sequence of events.

1 • Some or all of the remaining instructions are not executed.

2 • The rescue block of the routine is executed.

3 • If the rescue block executes a Retry, the body of the routine is executed again. This terminates processing of the current exception. Any new triggering of an exception is a new occurrence, which will (recursively) be handled according to the present semantics.

4 • If the rescue block is executed to the end without executing a Retry, this terminates the processing of the current exception and the current execution of *r*, causing a **failure** of that execution. If there is a calling routine, this failure triggers an exception in the calling routine, which will be handled (recursively) according to the same semantics. If there is no calling routine, *r* is the root's creation procedure; its execution will terminate.

The definition mentions that it applies only to ~~a routine~~ which is neither ignored nor continued. This corresponds to two facilities provided through features of the Kernel Library class *EXCEPTIONS*, implementing the false alarm response:

- You may specify that a certain type exception must be altogether ignored.
- You may specify that a certain type of exception must cause execution of a designated procedure and then continuation.

15.10 EXCEPTION CORRECTNESS

Definition: exception- correct

A routine *r* of a class *C* is **exception-correct** if and only if, for every branch *b* of its rescue block:

1 • If *b* ends with a Retry: {**true**} *b* $\{INV_C \wedge pre_r\}$

2 • If *b* does not end with a Retry: {**true**} *b* $\{INV_C\}$

In this rule, INV_C is the invariant of class *C* and pre_r is the precondition of *r*.

15 Exceptions

15.2 WHAT IS AN EXCEPTION?

Definitions: failure, exception

Under certain circumstances, the execution of a construct (such as an instruction) may be unable to terminate as you normally expect it to. The execution is then said to result in a failure.

If a routine executes a component and that component fails, this will prevent the routine's execution from proceeding as planned; such an event is called an exception.

15.3 EXCEPTION HANDLING POLICY

Only three possibilities make sense for handling an exception:

- A favorable albeit unlikely case is one in which the exception was in fact not justified. This is called the **false alarm**.
- When writing the component, you may have anticipated the possibility of an exception, and provided for an alternative way to fulfil the contract. Then the execution will try that alternative. This case is called **resumption**.
- If you have no way of fulfilling the contract, then you should try to return the objects involved into an acceptable state, and signal your failure to the client. This is called **organized panic**.

Definition: recipient

Any execution of a software component is part of the execution of a call to a certain routine, known as the *current routine*. If the component's execution fails, this will trigger an exception in the current routine, which becomes the recipient of the exception. Depending on how the software has been written, the exception will be handled through one of the three techniques listed above.

15.8 SYNTAX AND VALIDITY OF THE EXCEPTION CONSTRUCTS

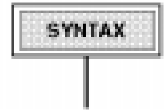
Rescue	$\triangleq$	**rescue** Compound
Retry	$\triangleq$	**retry**

VXRC

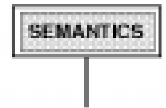
It is valid for a Routine to include a Rescue clause if and only if its Routine_body is of the Internal form.

computation unchanged if the Boolean_expression of its Exit clause evaluates to false; otherwise, it is the effect of executing the Compound clause, followed (recursively) by the effect of executing the Loop_body again in the resulting state.

14.8 THE DEBUG INSTRUCTION



Debug	$\triangleq$	**debug** [Debug_keys] Compound **end**
Debug_keys	$\triangleq$	"(" Debug_key_list ")"
Debug_key_list	$\triangleq$	{Debug_key "," ...}
Debug_key	$\triangleq$	Manifest_string



The effect of a Debug instruction depends on the mode that has been chosen for the enclosing class:

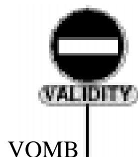
- If the *debug* option is on generally, or if the instruction includes a Debug_key_list and the option is on for at least one Debug_key in the list, the effect of the Debug instruction is the same as that of its Compound.
- Otherwise the effect is that of a Null instruction.

Character_interval	≜	Character_constant ".." Character_constant

Definition: unfolded form of a ~~multi_branch~~

To discuss the constraints and the semantics, it is convenient to consider the unfolded form of a Multi_branch. To obtain it, just replace any integer or character Interval, in the Choices of a When_part, by a Choices list made up of all constants between the interval's bounds, or empty if the second bound is smaller than the first. Integer order is used for an Integer_interval, and character code order for a Character_interval.

Multi_branch rule



A Multi_branch instruction is valid if and only if its unfolded form satisfies the following conditions.

1 • The inspect expression is of type *INTEGER* or *CHARACTER*.

2 • Any inspect constant (any value in one of the Choices parts) is a Constant of the same type as the inspect expression.

3 • Any two non-Unique inspect constants have different values.

4 • Any two Unique inspect constants have different names.

5 • If any inspect constant is Unique, then every other inspect constant in the instruction is either Unique or has a negative or zero value.

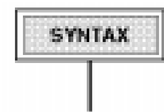
6 • All Unique inspect constants, if any, have the same class of origin (the enclosing class or a proper ancestor).

The effect of executing a Multi_branch instruction is defined as the effect of executing its unfolded form, as follows. The value of the inspect expression is computed. Because of the above validity constraint, that value may be equal to at most one of the inspect constants. If there is indeed one such constant, the effect of the Multi_branch is that of the Compound appearing after the **then** in the When_part of the matching inspect constant. If there is no such constant:

1 • If the Else_part is present, the effect of the Compound is that of the Compound appearing in the Else_part.

2 • Otherwise an exception is triggered and the current routine execution fails.

14.7 LOOP



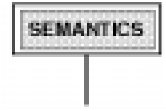
Initialization	≜	**from** Compound
Loop_body	≜	Exit **loop** Compound
Exit	≜	**until** Boolean_expression

The effect of a Loop is the effect of executing its Initialization followed by the effect of executing its Loop_body. The effect of executing an Initialization clause is the effect of executing its Compound. The effect of executing a Loop_body is to leave the state of the

Definition: prevail immediately

If the value of $condition_1$ is true when the instruction is executed, then the Conditional is said to prevail immediately.

Finally, we may consider that every Conditional has an Else_part if we understand an empty Else_part to stand for one with an empty Compound.



With these conventions, the effect of a Conditional may be defined as follows. If the Conditional prevails immediately, then its effect is that of its $compound_1$ part, as defined above. Otherwise:

- If it has a secondary part, the effect of the entire Conditional is (recursively) the effect of the secondary part.
- If it has no secondary part, its effect is that of the (possibly empty) Compound in its Else part.

14.5 MULTI-BRANCH CHOICE

Definition: inspect expression

A Multi_branch instruction contains a Expression, called the inspect expression, appearing after the keyword *inspect*. The inspect expression, *last_input* in the example, may only be of type *INTEGER* or, as here, *CHARACTER*. It includes one or more When_part, each of which indicates a list of one or more Choice, separated by commas, and a Compound to be executed when the value of the Expression is one of the given Choice values.

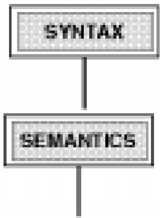
Definition: inspect constant

Every Choice specifies one or more values, called inspect constants. More precisely, a Choice is either a single constant (Manifest_constant or constant attribute) or an interval of consecutive constants yielding all the interval's elements as inspect constants. If present, the instruction's optional Else_part is executed when the inspect expression is not equal to any of the inspect constants.

Multi_branch	$\triangleq$	**inspect** Expression
		[When_part_list] [Else_part] **end**
When_part_list	$\triangleq$	**when** {When_part **when** ...}$^+$
When_part	$\triangleq$	Choices **then** Compound
Choices	$\triangleq$	{Choice "," ...}
Choice	$\triangleq$	Constant | Interval
Interval	$\triangleq$	Integer_interval | Character_interval
Integer_interval	$\triangleq$	Integer_constant "••" Integer_constant

14 Control structures

14.2 COMPOUND

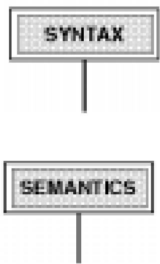


Compound	$\triangleq$	{Instruction ";" ...}

The effect of executing a Compound may be defined as follows.

- If the Compound has zero instructions, the effect is to leave the state of the computation unchanged.
- If the Compound has one or more instructions, its effect is that of executing the first instruction of the Compound and then (recursively) to execute the Compound obtained by removing the first instruction.

14.3 NULL INSTRUCTION



Specimens of the null instruction are empty.

The effect of the null instruction is to leave the state of the computation unchanged.

14.4 CONDITIONAL

Conditional	$\triangleq$	**if** Then_part_list [Else_part] **end**
Then_part_list	$\triangleq$	{Then_part **elseif** ...}$^+$
Then_part	$\triangleq$	Boolean_expression **then** Compound
Else_part	$\triangleq$	**else** Compound

To define precisely the semantics of this construct, a few auxiliary notions are useful. As the syntactical specification shows, a Conditional begins with

if $condition_1$ **then** $compound_1$

where $condition_1$ is a boolean expression and $compound_1$ is a Compound.

Definition: secondary part

The remaining part may optionally begin with **elseif**. If so, replacing the first **elseif** by **if** would transform the remaining part into a new, syntactically correct, Conditional; such an instruction is called the secondary part of the enclosing Conditional.

The final part, also optional, is of the form **else** $compound_n$.

13.11 BIT TYPES

The possible direct conformance cases involving a Bit_type are the following for any positive integers *N* and *P*:

1 • *BIT N* conforms directly to *ANY.*

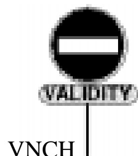
2 • *BIT N* conforms directly to *BIT P* for $N \leq$ P.

Other than implied by these rules, no type conforms directly to a Bit_type, and a Bit_type conforms directly to no type.

13.7 FORMAL GENERIC PARAMETERS

Let *G* be a formal generic parameter of a class, which in the class may be used as a type of the Formal_generic_name category. No type conforms directly to *G*. If *G* is not constrained, it conforms directly to the type *ANY* (based on the corresponding universal class) and to no other type. If *G* is constrained by *CT*, *G* conforms directly to *CT* and to no other type.

13.8 ANCHORED TYPES



In a class *C*, type **like** *Current* conforms directly to its base type *CT*, where *CT* is *C* followed by its Formal_generic_list, if any, with any Constraint removed.

Type **like** *anchor*, where *anchor* is a feature of *C* or a formal argument of a routine of *C*, conforms directly to the type of *anchor* in *C*.

An anchored type conforms directly to no type other than implied by these rules. No type conforms directly to an anchored type.

13.9 EXPRESSION CONFORMANCE

An expression *v* of type *VT* conforms to an expression *t* of type *TT* if and only if they satisfy any one of the following four conditions.

1 • *VT* conforms to *TT*.

2 • *TT* is **like** *v* (*v* in this case must be an entity).

3 • *VT* and *TT* are both of the form **like** *x* for the same *x*.

4 • *TT* is **like** *x* where *x* is a formal argument to a routine *r*, *v* is an actual argument in a call to *r*, and *VT* conforms to the type of the actual argument corresponding to *x* in the call.

5 • *v* is a call to some function *f* of type **like** *x* where *x* is a formal argument of *f*, and the type of the actual argument corresponding to *x* in the call conforms to *TT*.

13.10 EXPANDED TYPES

Definition: heavier arithmetic type

Any arithmetic type conforms to heavier ones, where *DOUBLE* is heavier than *REAL* and *INTEGER*, and *REAL* is heavier than *INTEGER*.

Let *T* be an expanded type other than a Bit_type. A type *U* conforms directly to *T* if and only if they satisfy any one of the following three conditions:

1 • *T* is of the form **expanded** *BT*, and *U* is *BT*.

2 • *T* is *REAL* and *U* is *INTEGER*.

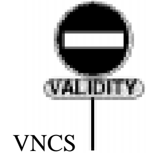
3 • *T* is *DOUBLE* and *U* is *REAL* or *INTEGER*.

In case 1 *T* also conforms directly to *U*.

An expanded type conforms directly to no type other than implied by this rule and the rules of 13.5 and 13.6.

13 Conformance

13.3 SIGNATURE CONFORMANCE

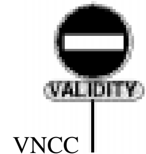


A signature $t = (<B_1, ..., B_n>, <S>)$ conforms to a signature $s = (<A_1, ..., A_n>, <R>)$ if and only if it satisfies the following conditions:

1 • Each of the two sequence components of *t* has the same number of elements as the corresponding component of *s*.

2 • Every type T_i in each of the two sequence components of *t* conforms to the corresponding type S_i in the corresponding component of *s*.

13.4 DIRECT AND INDIRECT CONFORMANCE



Let *T* and *V* be two types. *V* **conforms to** *T* if and only if one of the following holds:

1 • *V* and *T* are identical.

2 • *V* conforms directly to *T*.

3 • *V* is *NONE* and *T* is a reference type.

4 • *V* is $B\ [Y_1,... Y_n]$ for some generic class *B*, *T* is $B\ [X_1,... X_n]$, and every one of the Y_i conforms (recursively) to the corresponding X_i.

5 • *T* is a reference type and, for some type *U*, *V* conforms to *U* and *U* conforms (recursively) to *T*.

13.5 CONFORMANCE TO A NON-GENERIC REFERENCE TYPE

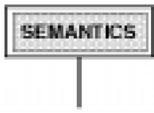
Let *CT* be a Class_type of base class *C*, and *BT* be a reference type whose base class *B* is not generic. *CT* conforms directly to *BT* if and only if the Inheritance clause of *C* lists *B* in one or more of its Parent items.

13.6 GENERICALLY DERIVED REFERENCE TYPES

Let *BT* be a generically derived reference type of base type $B\ [X_1,... X_n]$ for some n >= 1, where the formal generic parameters of *B* are $G_1,... G_n$. Let *CT* be a Class_type of base class *C* different from *B*. To determine whether *CT* conforms directly to *BT*, define the substitution σas follows:

- If *CT* is non-generic, σ is the identity substitution.
- If *CT* is a a generically derived type, of the form $C\ [Y_1,... Y_m]$, and class *C* is declared with formal generic parameters $H_1,... H_m$, then σ applied to any of the H_i (for $1 \leq i \leq m$) is Y_i, and σ applied to any other element is the element itself.

Then *CT* conforms directly to *BT* if and only if the Inheritance clause of *C* lists $B\ [Z_1,... Z_n]$ as one of its Parent items and, for every *j* such that ~~$1 \leq j \leq$~~ *n*, applying substitution σ to Z_j yields X_j.



The base type *BT* of an anchored type **like** *anchor* appearing in a class *C* is determined as follows:

Definition: base class, base type (anchored)

1 • If *anchor* is the final name of some feature of *C*, then *BT* is the declared type of that feature in *C*.

2 • If *anchor* is a formal routine argument, then *BT* is the type declared for that argument in the Formal_arguments list.

3 • If *anchor* is *Current*, then *BT* is *C* followed by its Formal_generics, if any, with any Constraint removed.

12.12 RULES ON EXPANDED TYPES

Definition: expanded type, reference type

A type *T* is expanded if and only if one of the following conditions holds:

1 • *T* is a Class_type whose base class *C* is an expanded class.

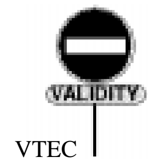
2 • *T* is of the form **expanded** *CT*. (As noted, it is redundant but not erroneous for the base class of *CT* to be an expanded class.)

3 • *T* is of the form *BIT* M for some non-negative integer M.

T is a reference type if it is not a Formal_generic_name and none of the above condition applies.

Expanded Type rule

It is valid to use an expanded type of base class *C* in the text of a class *B* if and only if it satisfies the following two conditions:



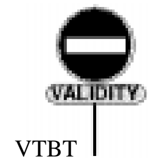
1 • *C* is not a deferred class.

2 • *C* either has no Creators part, or has a Creators part containing exactly one creation procedure, with no argument, available to *B* for creation.

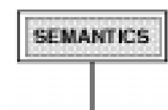
Any entity declared of an expanded type has run-time values which are instances of the corresponding base type.

12.14 *BIT* TYPES

A Bit_type declaration is valid if and only if its Constant is of type *INTEGER*, and has a positive value.



The possible values of an entity declared as *BIT N* for some *N* are bit sequences of exactly *N* bits.



12.15 ANCHORED TYPES

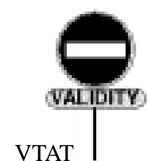
Definition: anchored type, anchor

An Anchored type is of the form

like *anchor*

where *anchor* is called the anchor of the type.

An anchored type of the form **like** *anchor* appearing in a class *C* is valid if and only if one of the following holds:



1 • *anchor* is the final name of an attribute or function of *C*, whose declared type is a non-Anchored reference type.

2 • The type appears in the text of a routine *r* of *C*, and *anchor* is a formal argument of *r*, whose declared type is a non-Anchored reference type.

3 • *anchor* is the reserved word *Current*.

Definition: generic class, generic derivation, non-generic

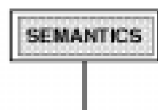
Any class declared with a non-empty Formal_generics part (constrained or not) is said to be a generic class. A generic class does not describe a type but a template for a set of possible types. To derive an actual type from this template, you must provide an Actual_generics list, whose elements are themselves types. The result is called a generic derivation.

Unconstrained Genericity rule

Let *CT* be a Class_type having a non-empty Actual_generics part, whose base class *C* is not a constrained generic class. *CT* is valid if and only if *C* satisfies the following two conditions:

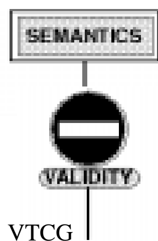
1 • *C* is a generic class.

2 • The number of Type components in *CT*'s Actual_generics list is the same as the number of Formal_generic parameters in the Formal_generic_list of *C*'s declaration.



A generically derived type is expanded if its base class is an expanded class; otherwise it is a reference type.

12.8 CONSTRAINED GENERICITY



The effect of a Constraint, if present, is to restrict allowable actual generic parameters to types that conform to the Class_type given.

Constrained Genericity rule

Let *C* be a constrained generic class. A Class_type *CT* having *C* as base class is valid if and only if *CT* satisfies the two conditions of the Unconstrained Genericity rule (~~VTUG, page \n(Qt)~~ and, in addition:

3 • For any Formal_generic parameter in the declaration of *C* having a constraint of the form –> *D*, the corresponding Type in the Actual_generics list of *CT* conforms to *D*.

12.9 USING FORMAL GENERIC PARAMETERS AS TYPES

Definition: base class, base type (constrained generic)

We consider the base type of a constrained generic parameter to be its constraining type, with the associated base class.

Definition: base class, base type (unconstrained generic)

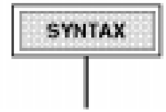
ANY serves as both the base type and the base class of any unconstrained Formal_generic_name.

12.11 CLASS TYPES EXPANDED

If *T* is a valid Class_type, generically derived or not, **expanded** *T* is a valid Class_type_expanded, and the possible values for entities of that type are instances of *T*.

12 Types

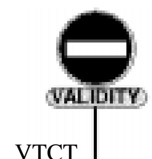
12.4 HOW TO DECLARE A TYPE



Type	≜	Class_type |
		Class_type_expanded |
		Formal_generic_name |
		Anchored |
		Bit_type
Class_type	≜	Class_name [*Actual_generics*]
Actual_generics	≜	"[" Type_list "]"
Type_list	≜	{Type "," ...}
Class_type_expanded	≜	**expanded** Class_type
Bit_type	≜	*BIT* Constant
Anchored	≜	**like** Anchor
Anchor	≜	Identifier | *Current*

12.5 BASE CLASS, BASE TYPE

Class Type rule



An Identifier *CC* is valid as the Class_name part of a Class_type if and only if it is the name of a class in the surrounding universe.

12.7 UNCONSTRAINED GENERICITY

Definition: base class, base type (class Type)

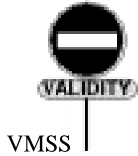
The base class of a generically derived type is the class used to derive it by providing actual generic parameters.

Definition: constrained, unconstrained generic

The syntax for Class_declaration includes an optional Constraint part after every formal generic parameter. If present, this part makes the parameter constrained; if not, the parameter is unconstrained. A generic class is constrained if it has at least one constrained parameter, unconstrained otherwise.

2 • If the Repeated Inheritance rule implies that *f* will be replicated in *D* and *f* is potentially ambiguous, then the Select subclause of exactly one of the Parent parts of *D* lists the corresponding version of *f*, under its final *D* name.

Select Subclause rule



VMSS

A Select subclause appearing in the Parent part for a class *B* in a class *D* is valid if and only if, for every Feature_name *fname* in its Feature_list, *fname* is the final name in *D* of a feature that has two or more potential versions in *D*, and *fname* appears only once in the Feature_list.

3 • If a class inherits two features as effective from different parents and they have the same name, the class must also (except under sharing for repeated inheritance) remove the name clash through renaming.

11.12 VERSIONS OF A FEATURE

Definition: potential version

Let *f* be a feature of a class *A* and *D* a descendant of *A*. A potential version of *f* in *D* is any inherited feature of *D* which is either:

- *f* itself.
- A feature resulting (recursively) from a redeclaration of a potential version of *f*.
- (Recursively) a potential version of a feature of which *f* is a redeclaration.
- A feature resulting (recursively) from a generic derivation of *A*.

Definition: version

Let *f* be a feature of a class *A* and *D* a descendant of *A*. The version of *f* in *D* is the feature *df* defined as follows:

1 • If *D* has only one potential version of *f*, then *df* is that feature.

2 • If *D* has two or more potential versions of *f*, the Repeated Inheritance Consistency constraint, seen below, states that exactly one of them must appear, under its final *D* name, as part of a Select clause in *D*; then *df* is that feature.

Definition: potentially ambiguous

Let *D* be a repeated descendant of a class *A*. A feature *f* of *A* is potentially ambiguous in *D* if and only if one of the following two conditions holds:

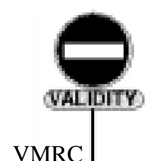
1 • *f* is an attribute.

2 • *D* has two or more potential versions of *f*.

11.13 THE REPEATED INHERITANCE CONSISTENCY CONSTRAINT

Select	$\triangleq$	**select** Feature_list

It is valid for a class *D* to be a repeated descendant of a class *A* if and only if *D* satisfies the following two conditions for every feature *f* of *A*:



VMRC

1 • If the Repeated Inheritance rule implies that *f* will be shared in *D*, then all the inherited versions of *f* are the same feature.

11.10 NAME CLASHES

Definition: name clash

A name clash occurs for a certain feature name *fname* in a class *C* if, for two different parents *A* and *B* of *C*, both *A* and *B* have a feature of name *fname*.

11.11 THE INHERITED FEATURES OF A CLASS

Definition: inherited features

Let *D* be a class. The list *inherited* of inherited features of *D* is obtained as follows. Let *precursors* be the list obtained by concatenating the lists of features of every parent of *D*; this list may contain duplicates in the case of repeated inheritance. Then *inherited* is obtained from *precursors* as follows:

1 • In list *precursors*, for any set of two or more elements representing features that are repeatedly inherited in *D* under the same name, so that the Repeated Inheritance rule yields sharing, keep only one of these elements. The Repeated Inheritance Consistency constraint (sharing case) indicates that these elements must all represent the same feature, so that it does not matter which one is kept.

2 • For every feature *f* in the resulting list, if *D* undefines *f*, replace *f* by a deferred feature with the same signature and specification.

3 • In the resulting list, for any set of deferred features with the same final name in *D*, keep only one of these features, with assertions joined as per the Join Semantics rule. (Keep the signature, which the Join rule requires to be the same for all the features involved.)

4 • In the resulting list, remove any deferred feature such that there is an effective feature with the same final name in the list. (This is the case in which a feature *f* inherited as effective effects one or more deferred features: of the whole group, only *f* remains.)

5 • Let *merged_features* be the resulting list. All its elements have different feature names; they are the inherited features of *D* in their parent form. From this list, produce a new one as follows: for any feature which *D* redeclares (by redefinition or effecting), replace the feature by the result of the redeclaration; keep any other feature as it is in *merged_features*.

6 • The result is the list *inherited* of inherited features of *D*.

VMFN

Feature Name rule

It is valid for a class *C* to introduce a feature with the Feature_name *fname*, or to inherit a feature under the final name *fname*, if and only if no other feature of *C* has that same name.

1 • A class may not introduce two different features, both deferred or both effective, with the same name.

2 • If a class introduces a feature with the same name as a feature it inherits in effective form, it must rename the inherited version.

11 Repeated inheritance

11.2 CASES OF REPEATED INHERITANCE

Definition: repeated inheritance, ancestor, descendant

Repeated inheritance occurs whenever (as a result of multiple inheritance) two or more of the ancestors of a class *D* have a common parent *A*. *D* is then called a repeated descendant of *A*, and *A* a repeated ancestor of *D*.

Definition: direct repeated inheritance

The simplest case, called direct repeated inheritance, corresponds to the following scheme (where *D* is a “repeated heir” of *A*):

class *D* **inherit**

A **rename** ... **redefine** ... **end**;

A **rename** ... **redefine** ... **end**

... Rest of class omitted ...

Definition: indirect repeated inheritance

The second case, indirect repeated inheritance, arises when one parent of *D* is a proper descendant of *A*, and one or more other parents are descendants of *A*. (Some of the paths may be direct.)

11.3 SHARING AND REPLICATION

Repeated Inheritance rule

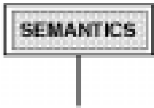
Let *D* be a class and B_1, ... B_n ($n \geq 2$) be parents of *D* having a common ancestor *A*. Let f_1, ... f_n be features of these respective parents, all having as their seed the same feature *f* of *A*. Then:

1 • Any subset of these features inherited by *D* under the same final name yields a single feature of *D*.

2 • Any two of these features inherited under a different name yield two features of *D*.

Definition: shared, replicated

Features will be said to be shared if case 1 of the Repeated Inheritance rule applies, and replicated if case 2 applies.



Join Semantics rule

Joining deferred features with the same final name yields a non-obsolete deferred feature defined as follows:

1 • Its name is the final name of all its precursors.

2 • Its signature is the precursors' signature, which the Join rule indicates must be the same for all precursors after possible redeclaration.

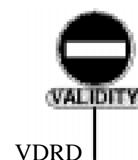
3 • Its precondition is the **or** of all the precursors' preconditions.

4 • Its postcondition is the **and** of all the precursors' postconditions.

5 • Its Header_comment is the concatenation of those of all precursors.

6 • It is not obsolete (even if some of the precursors are obsolete).

Redeclaration rule



Let *C* be a class and *g* a feature of *C*. It is valid for *g* to be a redeclaration of a feature *f* inherited from a parent *B* of *C* if and only if the following conditions are satisfied.

1 • No effective feature of *C* other than *f* and *g* has the same final name.

2 • The signature of *g* conforms to the signature of *f*.

3 • If *g* is a routine, its Precondition, if any, begins with **require else** (not just **require**), and its Postcondition, if any, begins with **ensure then** (not just **ensure**).

4 • If the redeclaration is a redefinition (rather than an effecting) the Redefine subclause of the Parent part for *B* lists the final name of *f* in its Feature_list.

5 • If *f* is inherited as effective, then *g* is also effective.

6 • If *f* is an attribute, *g* is an attribute, *f* and *g* are both variable, and their types are either both expanded or both non-expanded.

7 • If either one of *f* and *g* is an External routine, so is the other.

Definition: declared type

Any feature or entity of a class *C* has a **declared type** as follows:

- For a feature which is immediate in *C* or redeclared in *C*, *dt* is the type given by the declaration or redeclaration.
- For an inherited feature which is not redeclared in *C*, *dt* is (recursively) the declared type of its precursors in the corresponding parents.
- For the predefined entity *Current*, *dt* is *C* with its formal generic parameters if any.
- For the predefined entity *Result*, appearing in a function, *dt* is the return type declared for the function.
- For any other entity *e*, *dt* is the type used in the declaration of *e*.

Definition: type of a feature

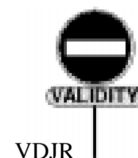
In this book, the “type” of a feature or entity, without further qualification, always means its declared type (rather than its base type).

10.23 RULES ON JOINING FEATURES

Definition: precursor (joined features)

A precursor of an inherited feature is a version of the feature in the parent from which it is inherited. Without the join mechanism there was just one precursor; but a feature which results from the join of two or more deferred features will have all of them as precursors.

Join rule



It is valid for a class *C* to inherit two different features as deferred under the same final name if and only if, after possible redeclaration in *C*, they have identical signatures.

1 • *f* is introduced in *C* as an attribute or a routine whose Routine_body is of the Effective form (that is to say, not the keyword **deferred** but beginning with **do**, **once** or **external**).

2 • *f* is an inherited feature, coming from a parent *B* of *C* where it is (recursively) effective, and *C* does not undefine it.

3 • Another feature of *C* with the same final name is (recursively) effective. That feature is then said to **effect** *f* in *C*.

A feature of *C* is a **deferred feature** of *C* if and only if it is not an effective feature of *C*.

Condition 3 defines the effecting case: an effective feature, which has the same final name as one or more deferred features, serves as effecting for all of them.

Definition: deferred, effective class

A class is **deferred** if it has at least one deferred feature. It is **effective** otherwise.

10.21 ORIGIN AND SEED

Definition: origin, seed

Every feature of a class *C* has a seed, which is a feature, and an origin, which is a class, defined as follows.

1 • Any immediate feature of *C* (in other words, any feature introduced in *C* rather than inherited) is its own seed, and has *C* as its origin.

2 • An inherited feature of *C* with two or more precursors, all of which have (recursively) the same seed *s*, also has *s* as its seed. (This is the case of sharing under repeated inheritance.)

3 • If *C* joins a set of inherited deferred features, yielding (as explained above) a single feature of *C* to which case 2 does not apply, that feature is its own seed and its origin is *C*.

4 • Any feature of *C* to which none of the previous cases applies is inherited, and has exactly one precursor; then its seed and origin are (recursively) the seed and origin of that precursor.

10.22 REDECLARATION RULES

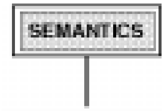
Definition: redeclaration, redefinition, effecting

A class *C* **redeclares** an inherited feature *f* if and only if one of the following two conditions holds:

- *C* contains a Feature_declaration for a feature *g* with the same final name as *f*.
- *C* inherits *f* as deferred, and inherits as effective another feature *g* with the same final name as *f*.

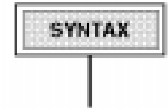
A **redefinition** is a redeclaration which is not an effecting.

10.17 THE JOIN MECHANISM



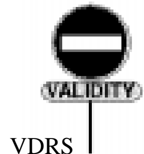
If *C* inherits and joins two or more deferred features, the net result for *C* is as if it had inherited a single deferred feature. In the absence of further action from *C*, that feature remains deferred. Of course, *C* may also provide an effective declaration for the feature, killing several abstract birds with one concrete stone by using a single redeclaration to effect several features inherited as deferred.

10.19 REDEFINITION AND UNDEFINITION RULES



Redefine	$\triangleq$	**redefine** Feature_list
Undefine	$\triangleq$	**undefine** Feature_list

Redefine Subclause rule



Consider a class *C* with a parent *B*. If a Parent part for *B* in *C* contains a Redefine subclause, that clause is valid if and only if every Feature_identifier *fname* that it lists (in its Feature_list) satisfies the following conditions:

1 • *fname* is the final name in *C* of a feature inherited from *B*.

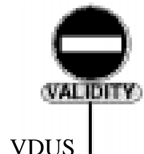
2 • That feature was not frozen, and was not a constant attribute.

3 • *fname* appears only once in the Feature_list.

4 • The Features part of *C* contains ~~oneFeature_declaration~~ for *fname*, which is a valid redeclaration, but not an effecting, of the original feature.

The effect of redefining a feature in a class is that any use of the feature in the class, its clients or (barring further redefinitions) its proper descendants will refer to the redefined version rather than the original.

Undefine Subclause rule



Consider a class *C* that inherits from a class *B*. If a Parent part for *B* in *C* contains an Undefine subclause, that clause is valid if and only if, for every Feature_identifier *fname* that it lists (in its Feature_list):

1 • *fname* is the final name in *C* of a feature inherited from *B*.

2 • That feature was not frozen, and was not an attribute.

3 • That feature was effective in *B*.

4 • *fname* appears only once in the Feature_list.

10.20 DEFERRED AND EFFECTIVE FEATURES AND CLASSES

Definition: effective feature, deferred feature, effecting

A feature *f* of a class *C* is said to be an **effective feature** of *C* if and only if any of the following conditions holds.

10 Feature adaptation

10.2 REDECLARING INHERITED FEATURES: WHY AND HOW

Definition: redeclaration

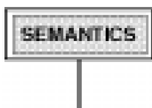
A class that contains a redefinition or effecting of an inherited feature will be said to redeclare that feature.

10.6 THE REDEFINITION CLAUSE

Definition: precursor

If a class inherits a feature from a parent, either keeping the feature unchanged or redefining it, the parent's version of the feature is called the precursor of the feature.

10.15 REDECLARATION AND ASSERTIONS



Consider a routine redeclaration and let pre_1, ... pre_n be the precursors' preconditions and $post_1$, ..., $post_n$ be the precursors' postconditions. Assume that new assertion clauses are present, of the form:

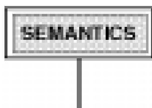
require else *alternative_precondition*

ensure then *extra_postcondition*

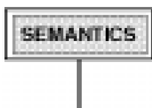
Then the redeclared routine will be considered to have the precondition and postcondition

alternative_precondition **or else** pre_1 **or else** ... **or else** pre_n

extra_postcondition **and then** $post_1$ **and then** ... **and then** $post_n$



If, in a routine redeclaration, the Precondition part is absent, the redeclared routine is considered to have **false** as its *alternative_precondition*; if the Postcondition part is absent, the redeclared routine is considered to have **true** as its *extra_postcondition*. Because of the rules of boolean algebra, the absence of one of these assertions means that the corresponding precursor assertion is kept as it was. (Or-ing a boolean value with **false**, or and-ing it with **true**, does not change the condition.)



For a declaration of an immediate feature of a class, the **require else** form of Precondition clause has the same meaning as if it were introduced by just **require**, and the **ensure then** form of Postcondition clause has the same meaning as if it were introduced by just **ensure**.

10.16 UNDEFINING A FEATURE

Definition: inherited as effective, as deferred

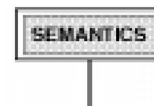
In the rest of this discussion, an inherited feature is said to be inherited as effective if it has at least one effective precursor and the corresponding Parent part does not undefine it. Otherwise the feature is inherited as deferred.

9.16 CORRECTNES S OF A CLASS

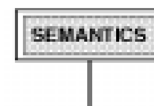
Definition: correctness (class)

A class is **correct** if and only if it is consistent and every routine of the class is check-correct, loop-correct and exception-correct.

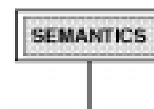
9.17 SEMANTICS OF ASSERTIONS



For a correct system, assertions, in all cases, will have no effect on the semantics of system execution (except through possible side effects of the functions called by assertions). For an incorrect system, the effect depends on compilation or execution options. Various options of the environment will make it possible to evaluate assertions. If an assertion evaluates to true, it has no further effect on the outcome of the computation. If it evaluates to false, it will trigger an exception, disrupting the normal flow of computation, as discussed in the chapter on exception handling.



An assertion violation detected as a result of enabling assertion monitoring at one of the above levels triggers an exception. An exception will also result, at level **loop** or higher, if a loop iteration fails to decrease the variant or gives it a negative value.



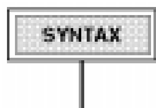
Here is the rule for determining the recipient of an exception resulting from an assertion violation:

1 • For postconditions, class invariants, loop invariants, variants and Check instructions, the recipient is the routine whose text contains the violated assertion or variant.

2 • For a violated precondition, the recipient is the calling routine. In this case no component of the routine's body is executed; the routine fails immediately, not performing any of its normal actions, and triggering an exception in the caller.

In this rule, INV_C is the invariant of C and, for any routine s, pre_s is the precondition of s, $post_s$ its postcondition, and do_s its body.

9.13 CHECK INSTRUCTIONS

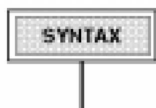


Check	$\triangleq$	**check** Assertion **end**

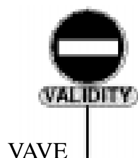
Definition: check-correct

An effective routine r is **check-correct** if, for every Check instruction c in r, any execution of c (as part of an execution of r) satisfies all its assertions.

9.14 LOOP INVARIANTS AND VARIANTS



Variant	$\triangleq$	**variant** [Tag_mark] Expression



A Variant is valid if and only if its Expression is of type *INTEGER*.

~~The invariant assertion *INV* of a loop must have the following two properties:~~

- ~~The loop's Initialization (**from** clause) ensures the truth of *INV*.~~
- ~~Any execution of the Loop_body, when started in a state that does not satisfy the Exit_condition, preserves the truth of *INV* (in other words, leaves *INV* true if it finds *INV* originally true).~~

Definition: loop-correct

A routine is **loop-correct** if every loop it contains satisfies the following four conditions:

1 • {**true**} *INIT* {*INV*}

2 • {**true**} *INIT* {*VAR* >= 0}

3 • {*INV* **and then not** *EXIT*} *BODY* {*INV*}

4 • {*INV* **and then not** *EXIT* **and then** (*VAR* = *v*)} *BODY* {$0 \leq VAR < v$}

where *INV* is the loop's invariant, *VAR* its variant, *INIT* its Initialization, *EXIT* its Exit condition, and *BODY* its Loop_body.

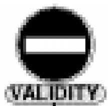
Definition: availability of an assertion clause

An Assertion_clause *a* of a routine Precondition or Postcondition is **available** to a class *B* if and only if all the entities involved in *a* are available to *B*, with the convention that formal arguments and *Result* are available to all classes.

9.9 OLD EXPRESSION

SYNTAX

Old	$\triangleq$ **old** Expression



VAOL

An Old expression of the form **old** *e*, where *e* is an expression of type *TE*, is valid if and only if it satisfies the following two conditions:

1 • It appears in a Postcondition clause of a Routine *r*.

2 • Transforming *r* into a function with result type *TE* (by adding a result type if *r* is procedure, or changing its result type if it is already a function) and replacing its entire Routine part by **do** *Result* := *e* **end** would yield a valid routine.

SEMANTICS

The value of an Old expression **old** *e* is defined only at the end of the execution of a call to *r*, just before the call returns; it is the result that would have been produced by evaluating *e* just before the call's execution began.

9.11 CLASS INVARIANTS

Definition: invariant of a class

The invariant of a class *C* is an assertion obtained by concatenating the following assertions (omitting any one which is absent or empty):

1 • The invariants of all parents (determined recursively through the present rule), in the order of the corresponding Parent clauses.

2 • The postconditions of any inherited functions which *C* redefines as an attribute, with every occurrence of *Result* replaced by the attribute's final name. (If there are two or more such redefinitions, include them in the order in which their new declarations appear in *C*.)

3 • The Assertion in *C*'s own Invariant clause, if any.

9.12 CONSISTENCY OF A CLASS

Definition: consistency

A class *C* is **consistent** if and only if it satisfies the following two conditions:

1 • For every creation procedure *p* of *C*: $\{pre_p\}\ do_p\ \{INV_C \wedge post_p\}$

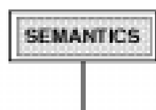
2 • For every routine *r* of *C* exported generally or selectively:

$$\{pre_r \wedge INV_C\}\ do_r\ \{post_r \wedge INV_C\}$$

9 Correctness

9.5 FORM OF ASSERTIONS

Precondition	$\triangleq$	**require** [**else**] Assertion
Postcondition	$\triangleq$	**ensure** [**then**] Assertion
Invariant	$\triangleq$	**invariant** Assertion
Assertion	$\triangleq$	{Assertion_clause ";" ...}
Assertion_clause	$\triangleq$	[Tag_mark]
		Unlabeled_assertion_clause
Unlabeled_assertion_clause	$\triangleq$	Boolean_expression | Comment
Tag_mark	$\triangleq$	Tag ":"
Tag	$\triangleq$	Identifier



In an Assertion, the semicolon separating each Assertion from the next has the same semantics as the **and then** infix boolean operator. This means that the order of the clauses may be meaningful:

1 • The value of an Assertion is true if and only if every Assertion_clause in the Assertion has value true.

2 • If an Assertion_clause has value false, the whole Assertion in which it appears has value false, even if the value of a subsequent clause is not defined.

9.7 THE SPECIFICATION OF A ROUTINE

Definition: specification, subspecification

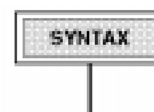
Let *pre* and *post* be the precondition and postcondition of a routine *rout*. The specification of *rout* is the pair of assertions <*pre*, *post*>.

A specification <*pre', post'*> is said to be a subspecification of <*pre, post*> if and only if *pre* implies *pre'* and *post'* implies *post*. Here “implies” is boolean implication.

9.8 CONSTRAINTS ON ROUTINE ASSERTIONS

A Precondition of a routine *r* of a class *C* is valid if and only if every feature whose final name appears in any Assertion_clause is available to every class to which *r* is available.

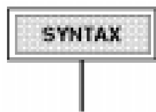
8.9 TYPES OF INSTRUCTIONS



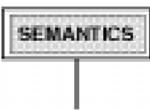
Instruction	$\triangleq$	Creation |
		Call |
		Assignment |
		Assignment_attempt |
		Conditional | Multi_branch | Loop |
		Debug | Check | Retry

2 • In the other cases (where the Routine_body is External or Deferred), there is neither a Local_declarations part nor a Rescue part.

8.6 ROUTINE BODY



<table>
<tr><td>Routine_body</td><td>≜</td><td>Effective | Deferred</td></tr>
<tr><td>Effective</td><td>≜</td><td>Internal | External</td></tr>
<tr><td>Internal</td><td>≜</td><td>Routine_mark Compound</td></tr>
<tr><td>Routine_mark</td><td>≜</td><td>do | once</td></tr>
<tr><td>Deferred</td><td>≜</td><td>deferred</td></tr>
</table>



The introductory keywords **do** or **once** of an Internal body correspond to different semantics for calls to the routine:

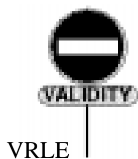
1 • For a **do** body, as indicated above, the initialization and body are executed anew on each call.

2 • If routine *o* of class *C* has a **once** body (*o* is then called a "once routine"), the initialization and body of *o* are executed only for the first call to *o* applied to an instance of *C* during any given session. For every subsequent call to *o* applied to an instance of *C* during the same session, the routine call has no effect; if the routine is a function, the value it returns is the same as the value returned by the first call.

8.7 LOCAL ENTITIES AND *Result*

<table>
<tr><td>Local_declarations</td><td>≜</td><td>local Entity_declaration_list</td></tr>
</table>

Local Entity rule



VRLE

Let *ld* be the Local_declarations part of a routine *r* in a class *C*. Let *locals* be the concatenation of every Identifier_list of every Entity_declaration_group in *ld*. Then *ld* is valid if and only if every Identifier *e* in *ld* satisfies the following two conditions:

1 • No feature of *C* has *e* as its final name.

2 • No formal argument of *r* has *e* as its Identifier.

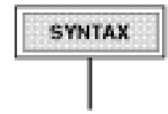
Definition: local entity

Most of the rules governing the validity and semantics of declared local entities also apply to a special predefined entity: *Result*, which may only appear in the Routine_ body or Postcondition of a function, and denotes the result to be returned by the function. Reflecting this similarity, this book uses the term "local entity" to cover *Result* as well as declared local entities.

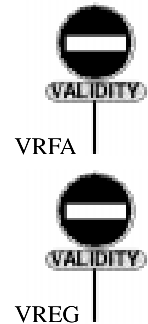
8 Routines

8.3 FORMAL ARGUMENTS

Formal_arguments	≜	"(" Entity_declaration_list ")"
Entity_declaration_list	≜	{Entity_declaration_group ";" ...}
Entity_declaration_group	≜	Identifier_list Type_mark
Identifier_list	≜	{Identifier "," ...}$^+$
Type_mark	≜	":" Type



Formal Argument rule

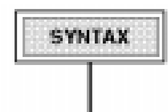


Let *fa* be the Formal_arguments part of a routine *r* in a class *C*. Let *formals* be the concatenation of every Identifier_list of every Entity_declaration_group in *fa*. Then *fa* is valid if and only if no Identifier *e* appearing in *formals* is the final name of a feature of *C*.

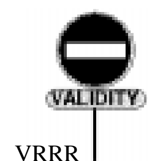
Let *el* be an Entity_declaration_list. Let *identifiers* be the concatenation of every Identifier_list of every Entity_declaration_group in *fa*. Then *el* is valid if and only if no Identifier appears more than once in the list *identifiers*.

8.5 ROUTINE STRUCTURE

Routine	≜	[Obsolete]
		[Header_comment]
		[Precondition]
		[Local_declarations]
		Routine_body
		[Postcondition]
		[Rescue]
		end ["--" Feature_name]



Routine rule



A Routine part of a routine declaration is valid if and only if one of the following conditions holds:

1 • Its Routine_body is an Internal body (beginning with **do** or **once**).

1 • At most one of the $feature_list_i$ is the keyword **all**.

2 • All the other $feature_list_i$ are lists of final names of features of *C* obtained from *B*.

3 • No final feature name appears twice in any such list, or appears in more than one list.

An immediate feature of a class has the following export status:

- If the Feature_clause which introduces it has no Clients part (that is to say, begins with the keyword **feature** with no further qualification), the feature is exported (generally available).
- If the Feature_clause which introduces it has a Clients part (that is to say, begins with **feature** {*A, B, C* ...}), the feature is selectively available to the descendants of the classes listed in that Clients part, and to these descendants only.

Definition: secret

If a Feature_clause has an empty Clients list, that is to say, begins with **feature** {}, then the features it introduces are secret.

If a non-redeclared inherited feature *f* has more than one precursor, it is available to all classes to which it would be available as a consequence of applying the preceding rule separately to each of its precursors.

7.14 DESCRIBING A CLASS FOR CLIENTS: THE SHORT FORM

Definition: short form, abstract form

The short form of a class, also called its abstract form, is a text which has the same structure as the class but does not include non-public elements. The short form is the one that should be used as interface documentation for the class.

7.15 THE FLAT-SHORT FORM

Definition: flat-short form

The flat-short form of a class is similar to the short form, but applies to the "reconstructed" full text of a class; you may view it as resulting from a shortening step that has been preceded by a "flattening" step, which expands the class text to unfold all the features obtained from proper ancestors, putting them at the same level as the immediate features of the class. Clearly, flattening must take both renaming and redefinition into account.

7.12 THE EXPORT STATUS OF FEATURES

Definition: exported, selectively available

The status of a feature of a class is one of the following:

1 • The feature may be available to all classes. Such a feature is said to be **exported**, ~~of~~ **generally available**.

2 • The feature may be available to specific classes only. In that case it is also available to the descendants of all these classes. Such a feature is said to be **selectively available** to the given classes and their descendants.

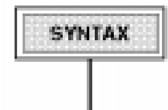
3 • The feature may be available to no classes. Then it is said to be **secret**.

Definition: available

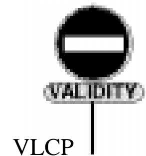
A feature of a class *S* is said to be **available** to a class *C* if and only if it is either selectively available to ~~*S*~~ or generally available.

7.13 ADAPTING THE EXPORT STATUS

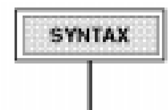
Clients	$\triangleq$	"{" Class_list "}"
Class_list	$\triangleq$	{Class_name "," ...}



A Clients part is valid if and only if every Class_name appearing in its Class_list is the name of a class in the surrounding universe.



New_exports	$\triangleq$	**export** New_export_list
New_export_list	$\triangleq$	{New_export_item ";" ...}
New_export_item	$\triangleq$	Clients Feature_set
Feature_set	$\triangleq$	Feature_list | **all**
Feature_list	$\triangleq$	{Feature_name "," ...}

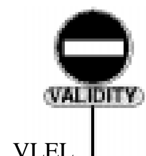


Export List rule

A New_exports parent appearing in class *C* in a Parent clause for a parent *B*, of the form

export {*class_list1*} *feature_list*$_1$; ... {*class_list*$_n$} *feature_list*$_n$

is valid if and only if (for *i* in the interval 1..*n*):



7 Clients and exports

7.3 CONVENTIONS

Definition: client

A class *C* is a **client** of *S* if some ancestor of *C* is a simple client, an expanded client or a generic client of *S*.

Definition: client relation between classes

A class *C* is a client of a class *B* if *C* is a client of a type whose base class is ~~*B*.~~

A class *C* is a **direct or indirect client** of a type *S* of base ~~type~~ *B* if there is a sequence of classes ~~C1 ==~~ A, C_2, ..., ~~Cn ==~~ *B* such that $n > 1$ and every C_i is a client of ~~C{i+1}~~ for ~~1 le i lt~~ *n*.

The "direct or indirect" forms of the simple client, expanded client and generic client relations are defined similarly.

7.4 SIMPLE CLIENTS

Definition: simple client

A class *C* is a **simple client** of a type *S* if some entity or expression of *C* is of type *S*.

7.5 EXPANDED CLIENTS

Definition: expanded client

A class *C* is an **expanded client** of a type *S* if *S* is an expanded type and some attribute of *C* is of type *S*.

Expanded Client rule

VLEC

It is valid for a class *C* to be an expanded client of a class *SC* if and only if *SC* is not a direct or indirect expanded client of *C*.

7.6 GENERIC CLIENTS

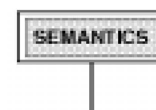
Definition: generic client, generic supplier

A class *C* is a **generic client** of a type *S* if for some generically derived type *T* of the form *B* [..., *S*, ...] one of the following holds:

1 • *C* is a client of *T*.

2 • One of the Parent clauses of *C*, or of a proper ancestor of *C*, lists *T* as parent.

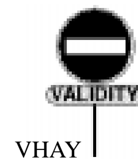
6.12 *ANY*



Any class other than *GENERAL* and *ANY* which does not include an explicitly written Inheritance clause is considered to have an implicit clause of the form

inherit *ANY*

6.15 PROVIDING YOUR OWN UNIVERSAL CLASS



VHAY

Whether you use the default *ANY* or another one, any system will need to have a class of name *ANY.* This is a constraint on any valid universe.

6.10 FEATURES AND THEIR NAMES

Definition: name of a feature in a class

Within the text of a class *C*, any feature *f* of *C* is accessible through a feature name, known as the name of *f* in *C*. As this expression suggests, the association between a feature and a feature name is not absolute but relative to a class. The same feature may well be denoted by different names in different classes.

Definition: original name

The **original name** of a feature is the name under which it is declared in its class of origin.

Definition: final name

Every feature *f* of a class *C* has a final name in *C*, defined as follows:

- If *f* is immediate in *C*, its final name is its original name
- If *f* is inherited, *f* is obtained from a feature of a parent *B* of *C*. Let *parent_name* be (recursively) the final name of that feature in *B*. Then:
- If the Parent clause for *B* in *C* contains a Rename_pair of the form **rename** *parent_name* **as** *new_name*, the final name of *f* in *C* is *new_name*. Otherwise, the final name is *parent_name*.

Definition: final name set

The final names of all the features of a class constitute the final name set of a class.

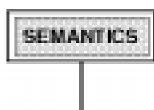
Definition: inherited name

The **inherited name** of a feature obtained from a feature *f* of a parent *B* is the final name of *f* in *B*.

Definition: name of a feature

In this book, references to the "name" of a feature, if not further qualified, always denote the final name.

6.11 INHERITANCE AND EXPANSION



The only consequence of the expansion status of a class is the semantics of entities of the corresponding types, such as *x* above. An expanded class may inherit from an non-expanded one, and conversely. The expansion status is not transmitted under inheritance; it is entirely determined by the presence or absence of the **expanded** mark in the class's own Class_header, not by any property of its parents.

Definition: ancestor, descendant

Class *A* is an **ancestor** of class *B* if and only if *A* is *B* itself or, recursively, an ancestor of one of *B*'s parents.

Class *B* is a **descendant** of class *A* if and only if *A* is an ancestor of *B*, in other words if *B* is *A* or (recursively) a descendant of one of its heirs.

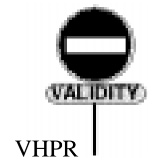
Definition: proper ancestor, descendant

The **proper ancestors** of a class *C* are its ancestors other than *C* itself. The **proper descendants** of a class *B* are its descendants other than *B* itself.

6.6 THE INHERITANCE STRUCTURE

Parent rule

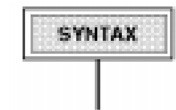
The Inheritance clause of a class *D* is valid if and only if it meets the following two conditions:



1 • In every Parent clause for a class *B*, *B* is not a descendant of *D*.

2 • If two or more Parent clauses are for classes which have a common ancestor *A*, *D* meets the conditions of the Repeated Inheritance Consistency constraint for *A*.

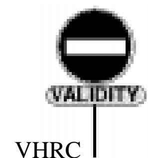
6.9 RENAMING



Rename	$\triangleq$	**rename** Rename_list
Rename_list	$\triangleq$	{Rename_pair "," ...}
Rename_pair	$\triangleq$	Feature_name **as** Feature_name

Rename Clause rule

A Rename_pair of the form *old_name* **as** *new_name*, appearing in the Rename subclause of the Parent clause for *B* in a class *C*, is valid if and only if it satisfies the following five conditions:



1 • *old_name* is the final name of a feature *f* of *B*.

2 • *old_name* does not appear as the first element of any other Rename_pair in the same Rename subclause.

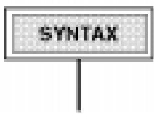
3 • *new_name* satisfies the Feature Name rule for *C*.

4 • If *new_name* is of the Prefix form, *f* is an attribute or a function with no argument.

5 • If *new_name* is of the Infix form, *f* is a function with one argument.

Renaming does not affect the semantics of an inherited feature, but simply gives it a new final name in an heir, as defined below.

6.3 FORM OF THE INHERITANCE PART



Inheritance	$\triangleq$	**inherit** Parent_list
.xk inherit		
Parent_list	$\triangleq$	{Parent ";" ...}
Parent	$\triangleq$	Class_type [Feature_adaptation]
Feature_adaptation	$\triangleq$	[Rename]
		[New_exports]
		[Undefine]
		[Redefine]
		[Select]
		end

Inheritance	$\triangleq$	**inherit** Parent_list
Parent_list	$\triangleq$	{Parent ";" ...}
Parent	$\triangleq$	Class_type [Feature_adaptation]
Feature_adaptation	$\triangleq$	[Rename]
		[New_exports]
		[Undefine]
		[Redefine]
		[Select]
		end

Definition: parent clause for a class

The Parent_list names a number of Parent clauses. Each Parent clause is relative to a Class_type, that is to say a class name *B* possibly followed by actual generic parameters (as in *B* [*T, U*]). *B* must be the name of a class in the universe to which the current class belongs. The clause is said to be a “Parent clause for *B*”.

6.4 RELATIONS INDUCED BY INHERITANCE

Definition: heir, parent

If class *C* has a Parent clause for *B*, then *C* is said to **inherit** from *B*; *B* is said to be a **parent** of *C*, and *C* is said to be an **heir** of *B*.

6 The inheritance relation

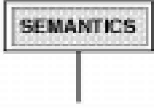
6.2 AN INHERITANCE PART

Definition: multiple, single inheritance

Multiple inheritance occurs as soon as there is more than one Parent clause (even if they all refer to the same parent class, a case called repeated inheritance and studied in chapter 11).

The special case is that of a multiple declaration introducing Unique constant attributes, which is covered by the Unique Declaration rule.

5.17 OBSOLETE FEATURES



Declaring a routine as Obsolete does not affect its semantics. But language processing tools, or at least some of them, should produce a warning when they process a client or descendant class that uses the routine. The warning should include the Message.

Definition: same feature name

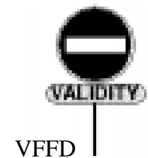
By convention, two feature names are the same if and only if either of the following conditions holds:

- They are both identifier features, with identical lower name.
- They are both operator features, both Prefix or both Infix, with identical lower names.

5.14 VALIDITY OF FEATURE DECLARATIONS

Feature Declaration rule

A Feature_declaration appearing in a class C, and whose New_feature_list contains one or more feature names f_1, ..., f_n, is valid if and only if it satisfies all of the following conditions:

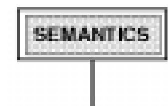


- Its Declaration_body describes a feature which, according to the rules of 5.11, is one of: variable attribute, constant attribute, procedure, function.
- None of the f_i has the same name as another feature introduced in C (in particular, f_i is not the same name as f_j for different i and j).
- If the name of any of the f_i is the same as the final name of any inherited feature, the Declaration_body satisfies the Redeclaration rule.
- If the Declaration_body describes a deferred feature, then none of the f_i is preceded by the keyword **frozen**.
- If any of the f_i is a Prefix name, the Declaration_body describes an attribute or a function with no argument.
- If any of the f_i is an Infix name, the Declaration_body describes a function with exactly one argument.
- If the Declaration_body describes a once function, the result type involves neither a Formal_generic_name nor an Anchored type.

5.16 SYNONYMS

Multiple Declaration rule

The semantics of a feature declaration applying to more than one feature name, as in



$f_1, f_2, \ldots, f_n$ *some_declaration_body*

is (except in one special case) defined as the semantics of the corresponding sequence of declarations naming only one feature each, and with identical declaration bodies, as in:

f_1 *some_declaration_body*;

f_2 *some_declaration_body*;

...

f_n *some_declaration_body*

5.12 THE SIGNATURE OF A FEATURE

Definition: signature

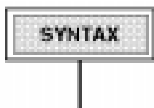
The **signature** of a feature *f* is a pair <*argument_types*, *result_type*> where both *argument_types* and *result_type* are sequences of types, defined as follows.

- If *f* is a routine, *argument_types* is the possibly empty sequence of the types of its arguments. If *f* is an attribute, *argument_types* is an empty sequence.
- If *f* is an attribute or a function, *result_type* is a one-element sequence, whose single element is the type of *f*. If *f* is a procedure, *f* is an empty sequence.

Definition: argument signature

The first component of a feature's signature, written *argument_types* in this definition, is called the argument signature of the feature. The argument signature gives the types of the feature's arguments; it is an empty sequence for attributes and for routines without arguments.

5.13 FEATURE NAMES



<table>
<tr><td>Feature_name</td><td>≜</td><td>Identifier | Prefix | Infix</td></tr>
<tr><td>Prefix</td><td>≜</td><td>prefix '"' Prefix_operator '"'</td></tr>
<tr><td>Infix</td><td>≜</td><td>infix '"' Infix_operator '"'</td></tr>
<tr><td>Prefix_operator</td><td>≜</td><td>Unary | Free_operator</td></tr>
<tr><td>Infix_operator</td><td>≜</td><td>Binary | Free_operator</td></tr>
</table>

Definition: identifier feature, operator feature

A feature declared with an identifier is called an identifier feature. A feature declared with an operator is called an operator feature.

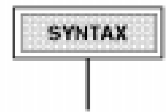
<table>
<tr><td>Unary</td><td>≜</td><td>~~not | "+" | "\-"~~not | "+" | "-"</td></tr>
<tr><td>Binary</td><td>≜</td><td>~~"+" | "–" | "*" | "^" |~~"+" | "–" | "*" | "/" |
"<" | ">" | "<=" | ">=" |
"//" | "\\" | "^" |
and | or | xor |
and then | or else | implies</td></tr>
</table>

A routine is either a **procedure** or a **function**:

- A procedure does not return a result; it may perform a number of operations, some of which may modify the instance to which the procedure is applied.
- A function returns a result and may also perform operations.

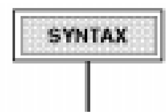
5.9 FEATURE DECLARATIONS: SYNTAX

Feature_declaration	≜	New_feature_list Declaration_body
Declaration_body	≜	[*Formal_arguments*]
		[Type_mark]
		[Constant_or_routine]
Constant_or_routine	≜	**is** Feature_value
Feature_value	≜	Manifest_constant | *Unique* | Routine



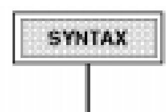
5.10 COMPONENTS OF A FEATURE DECLARATION

New_feature_list	≜	{New_feature "," ...} $^+$
New_feature	≜	[**frozen**] Feature_name



5.11 HOW TO RECOGNIZE FEATURES

A Feature_declaration is a **variable attribute** declaration if and only if it satisfies the following conditions:

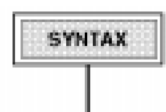


- There is no Formal_arguments part.
- There is a Type_mark part.
- There is no Constant_or_routine part.

A Feature_declaration is a **constant attribute** declaration if and only if it satisfies the following conditions:

- There is no Formal_arguments part.
- There is a Type_mark part.
- There is a Constant_or_routine part, which contains either a Manifest_constant or a Unique.

A Feature_declaration is a **routine** declaration if and only if it satisfies the following condition:



- There is a Constant_or_routine part, whose Feature_value is a Routine.

In this case the Formal_arguments and Type_mark parts may or may not be present. If the Type_mark is present, the declaration describes a **function**; otherwise it describes a **procedure**.

5 Features

5.3 IMMEDIATE AND INHERITED FEATURES

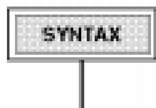
Definitions: features of a class, inherited, origin, introduced

The features of a class *C* include its inherited features and its immediate features, defined as follows:

1 • The features obtained by *C* from its parents, if any, are its inherited features.

2 • In the Features part of *C*, consider a declaration describing a feature *f*. If *f* is inherited, the declaration is in fact a **redeclaration** of *f*, giving *f* new properties in *C*. If this is not the case, *f* is a new feature, said to be immediate in *C*. *C* is then the **class of origin** (or simply "origin") of *f*, which is also said to be **introduced in** *C*.

5.6 FEATURES PART: SYNTAX



Features	$\triangleq$	**feature** {*Feature_clause* **feature** ...} $^{+}$
Feature_clause	$\triangleq$	[Clients]
		[Header_comment]
		Feature_declaration_list
Feature_declaration_list	$\triangleq$	{Feature_declaration ";" ...}
Header_comment	$\triangleq$	Comment

5.7 FORMS OF FEATURES

Every feature of a class is either an **attribute** or a **routine**.

Definition: field

By introducing an attribute in a class, you specify that at run-time every instance of the class will possess a certain value, or field, corresponding to the attribute.

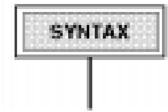
An attribute is either **variable** or **constant**:

- If an attribute is variable, the corresponding fields may be different for various instances of the class and may change at run-time. As a consequence, the actual values must be stored in the representation of each instance.
- If an attribute is constant, the corresponding field is the same value for all instances, and may not change at run-time. This value appears in the class as part of the attribute declaration.

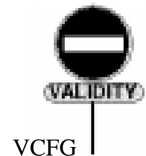
By introducing a routine in a class, you specify that a certain computation (an algorithm) must be applicable to instances of the class.

4.9 FORMAL GENERIC PARAMETERS

Formal_generics	$\triangleq$	"[" Formal_generic_list "]"
Formal_generic_list	$\triangleq$	{Formal_generic ","...}
Formal_generic	$\triangleq$	Formal_generic_name [Constraint]
Formal_generic_name	$\triangleq$	Identifier
Constraint	$\triangleq$	"–>" Class_type



A Formal_generics part of a Class_declaration is valid if and only if every Formal_generic_name *G* appearing in it satisfies the following three conditions:



VCFG

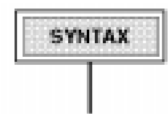
1 • *G* is different from the name of any class in the surrounding universe.

2 • *G* is different from any other Formal_generic_name appearing in the same Formal_generics_part.

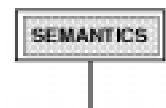
3 • If a Constraint is given, it does not involve any types other than class names and formal generic parameters other than *G* itself.

4.10 OBSOLETE CLAUSE

Obsolete	$\triangleq$	**obsolete** Message
Message	$\triangleq$	Manifest_string

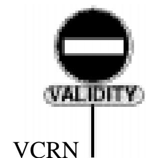


Declaring a class as Obsolete does not affect its semantics. But some language processing tools should produce a warning when they process a class that relies, as client or descendant, on an obsolete class. The warning should include the Message.



4.11 ENDING COMMENT

If present, the ending comment must repeat the Class_name given at the head of the class.



VCRN

4.5 CLASS TEXT STRUCTURE

Class_declaration	$\triangleq$	[Indexing]
		Class_header
		[Formal_generics]
		[Obsolete]
		[Inheritance]
		[Creators]
		[Features]
		[Invariant]
		end ["--" class Class_name]

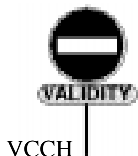
4.7 INDEXING A CLASS

Indexing	$\triangleq$	**indexing** Index_list
Index_list	$\triangleq$	{Index_clause ";" ...}
Index_clause	$\triangleq$	[*Index*] Index_terms
Index	$\triangleq$	Identifier ":"
Index_terms	$\triangleq$	{Index_value "," ...} $^{+}$
Index_value	$\triangleq$	Identifier | Manifest_constant

The Indexing clause has no effect on the execution semantics.

4.8 CLASS HEADER

Class_header	$\triangleq$	[Header_mark] **class** Class_name
Header_mark	$\triangleq$	**deferred** | **expanded**
Class_name	$\triangleq$	Identifier



A Class_header appearing in the text of a class *C* is valid if and only if it satisfies either of the following two conditions:

1 • There is no header mark of the **deferred** form, and *C* is effective.

2 • There is a Header_mark of the **deferred** form, and *C* is deferred.

4 Classes

4.2 OBJECTS

Definitions: object, instance

Viewed as a type, a class describes the properties of a set of possible data structures, or objects, which may exist during the execution of a system that includes the class; these objects are called the instances of the class.

4.3 FEATURES

Definitions: feature, attribute, routine

Viewed as a module, a class introduces, through its class text, a set of **features**. Some features, called **attributes**, represent fields of the class's direct instances; others, called **routines**, represent computations applicable to those ~~instances. .lY~~

Since there is no other modular facility than the class, building a software system in Eiffel means analyzing the types of objects the system will manipulate, and writing a class for each of these types.

case-independent names, even if the names are written with different case conventions in the class texts.

3.4 SYSTEMS

Definition: System, Root

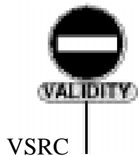
A **system** is a set of classes, one of which has been designated as the **root** of the system, such that all the classes on which the root depends belong to the system.

Definition: Dependency

Here a class *C* is said to **depend** on a class *A* if one of the following holds:

- *C* is an heir of *A*.
- *C* is a client of *A*.
- Recursively, there is a class *B* such that *C* depends on *B* and *B* depends on *A*.

Root Class rule



A class *C* may be used as root of a system if and only if it satisfies the following three conditions:

1 • *C* is not generic.

2 • *C* is not deferred.

3 • Any creation procedure of *C* has either no formal argument, or a single formal argument of type *ARRAY* [*STRING*].

3.5 CLUSTERS

Definition: Cluster

A cluster is a set of classes, all with different names.

No two classes in a given cluster may have the same class name.

3.6 UNIVERSES

Definition: Universe

A universe is a set of clusters.

3 The architecture of Eiffel software

3.1 OVERVIEW

The constituents of Eiffel software are called *classes*. By extracting classes from a given *universe*, you may assemble them into executable *systems*. To keep your classes and your development organized, it is convenient to group classes into *clusters*.

These four concepts provide the basis for structuring Eiffel software:

- A class is a modular unit.
- A system results from the assembly of one or more classes to produce an executable unit.
- A cluster is a set of related classes.
- A universe is a set of clusters, out of which developers will pick classes to build systems.

Of these, only "class", describing the basic building blocks, corresponds directly to a construct of the language. To build systems out of classes, you will use not a language mechanism, but tools of the supporting environment. As to clusters and universes, they are not language constructs but mechanisms for grouping and storing classes using the facilities provided by the underlying operating system, such as files and directories.

3.3 CLASS TEXTS AND CLASS NAMES

Every class has a name, such as *DOCUMENT* or *PARAGRAPH*, and a class text which describes the features of the class and its other properties.

For class names, as for all uses of identifiers, letter case is not significant: identifiers such as *DOCUMENT*, *document* and even *dOcUmEnT* have the same semantics when viewed as class names.

~~**Definition: Upper Name**~~

The standard recommended style in Eiffel texts is to write all class names using exclusively the upper name of the ~~class, that is to say the name all in upper case (such as *DOCUMENT*).~~

Definition: Class Name

The term "class name" as used in this book denotes the upper name of a class. In particular, two classes are said to have the same class name if they have the same

2.11 REQUIREMENTS ON LANGUAGE PROCESSING TOOLS

The definition of Eiffel syntax, validity and semantics contained in this book is also a specification of certain aspects of the corresponding language processing tools.

Not all aspects apply to all language processing tools.

A language processing tool that processes software components at a certain level (syntax, validity, semantics) is not required to perform the tasks associated with that level on components which do not ~~not~~ satisfy the requirements at the previous levels.

In almost all cases, authors of tools should follow a stricter guideline and make sure that their tools **reject** any input that does not satisfy the rules applying to the earlier levels. Such rejection should include a clear error message. For syntax, the message should identify the production which is not properly observed; for validity it should give the code of the violated validity constraint (and the clause number for constraints divided into clauses).

Two special considerations may justify occasional departures from this general obligation of rejection:

1 • A semantic tool may be able to process valid parts of a text, even if other parts are invalid. For example, a compiler may generate code for some valid classes in a system, rejecting classes which are not valid.

2 • A tool author may have a particular reason for providing a tool or tool option which accepts input violating a specific validity constraint. A possible application would be for a prototyping mode which attempts to execute incomplete systems, or skips certain checks. Such tool variants are outside of the semantics of Eiffel proper and should be clearly labeled as such, reminding developers that acceptance of an input text provides no guarantee that the text satisfies the full language rules.

One final note, intended for implementors of Eiffel, and regarding what they might *not* find here. Although this book goes to great lengths to include every relevant validity and semantic property, it may of course have left an occasional one out. Such an oversight might be a case of *incompleteness* (a missing validity constraint or semantic specification) or *inconsistency* (ambiguous or contradictory answers).

If you run into such a case while trying to produce language processing tools, please contact the language committee of NICE by sending electronic mail to <*language-chair @nice.twr.com*>,

Every validity constraint has a four-character code beginning with V (shown as VVVV in the above fictitious example).

2.7 INTERPRETING THE CONSTRAINTS

General Validity rule

Every validity constraint relative to a construct is considered to include an implicit supplementary condition stating that every component of the construct satisfies every validity constraint applicable to the component.

2.8 SEMANTICS

A construct specimen which is syntactically legal and valid has an associated semantics, specifying its run-time effect in a system in which the specimen appears. That semantics may include executing certain actions, producing a value, or both. It is defined by the paragraphs. For specimens made of further components, the specification usually refers recursively to the components' own semantics.

It is important to remember that the SEMANTICS paragraphs only apply to valid specimens. In many cases, the semantic rules would not even make sense otherwise. Clearly, attempting to describe the effect of an invalid component would be useless.

2.9 CORRECTNESS

Validity is only a structural property; valid Eiffel software is not guaranteed to perform according to any expected behavior. In fact, execution of valid software may lead to non-termination, or to exceptions and eventual failure.

For a valid component, then, we need a more advanced criterion: its ability to operate properly at run-time. This is called *correctness* and is a more elusive aim than validity, since it involves semantic properties.

2.10 THE CONTEXT OF EXECUTING SYSTEMS

Definitions: run time, machine, platform, language processing tool

The following terminology will serve to discuss the context of system execution:

- *Run time* is the period during which a system is executed.
- The *machine* is the combination of hardware (one or more computers) and operating system which makes it possible to execute systems.
- The machine type, that is to say a certain kind of computer and a certain operating system, is called a *platform*.
- To make the text of an Eiffel system executable by a machine, you will need software tools such as compilers and interpreters, for which this book will use the term *language processing tool*, general enough to cover various implementation techniques.

2.4 PRODUCTIONS

Definition: production

To understand a non-terminal, you need a formal description of the structure of its specimens. Such a description is called the **production** for the construct.

A production has the form

Construct $\triangleq$ right-hand-side

Every non-terminal construct appears on the left-hand side of exactly one such production. The symbol$\triangleq$ means “is defined as”.

The right-hand-side of the production describes the structure of specimens of the left-hand-side construct. Three forms of right-hand-side are available:

- Aggregate, describing a construct made of a fixed number of parts (some of which may be optional) to be concatenated in a given order.
- Choice, describing a construct having a number of given variants.
- Repetition, describing a construct made of a variable number of parts, which are all specimens of a given construct.

An aggregate right-hand-side is a non-empty sequence of constructs, some of which may be in square brackets [ ] to indicate optional parts.

A choice right-hand side is a non-empty sequence of constructs separated by vertical bars | .

A repetition right-hand side is of one of the two forms

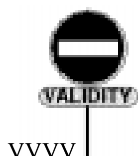
{Construct § ...} {Construct § ...} $^{+}$

where §, the separator, is some construct — usually, but not necessarily, terminal. Appearing in a production for a left-hand side construct R, this states that a specimen of R consists of zero or more specimens of B, separated, if more than one, by the separator §. In the first form, without a +, specimens of R may be empty; in the second form, with a +, they must include at least one B.

2.6 VALIDITY

The productions and other elements labeled , as described so far, specify the structure of constructs.

Definition: validity constraint, valid



Supplementary requirements on the syntactically well-formed specimens of a construct are called **validity constraints** on the construct. Paragraphs introducing them are labeled by the VALIDITY road sign.

A specimen which follows the syntactic rules and satisfies the constraints will be accepted by the language processing tools of any Eiffel environment and is said to be **valid**.

2 Syntax, validity and semantics

2.2 SYNTAX, COMPONENTS, SPECIMENS AND CONSTRUCTS

Eiffel's syntax is the set of rules describing the structure of class texts. It covers neither limitations on valid texts (described by validity constraints) nor the execution-time meaning or effect of these texts (described by semantic rules).

Definition: component

Any class text, or syntactically meaningful part of that text, such as an instruction, an expression or an identifier, is called a **component**.

Definition: construct, specimen

The structure of any components is described by a **construct**. A component conforming to a certain construct is called a **specimen** of that construct. For example, the construct Class describes the structure of class texts; any particular class text, built according to the rules given in this book, is a specimen of this construct.

All constructs have names beginning with a capital letter and written in the default (roman) font. Each appears in the index with a reference to the page of its syntactical definition.

An important convention will simplify the discussions: the phrase "an X", where X is the name of a construct, serves as a shorthand for "a specimen of construct X". For example, "a Class" means "a specimen of construct Class", in other words a text built according to the syntactical specification of construct Class.

2.3 TERMINALS, NON-TERMINALS AND TOKENS

Every construct is either a "terminal" or a "non-terminal" as defined next.

Definitions: Terminal, Non-terminal, Token

Specimens of a terminal construct have no further syntactical structure. Examples include reserved words (such as **class**, *Result* etc.), constants such as integers, and identifiers used to denote classes, features and entities. In contrast, the specimens of a non-terminal construct are defined in terms of other constructs.

Definition: token, lexical component

The specimens of terminal constructs are called **tokens** or **lexical components**. They form the basic vocabulary out of which you may build more complex texts — specimens of non-terminals.

A preliminary standard document for the Kernel Library, known as ~~the PELKS (Proposed~~ *Eiffel Library Kernel* ~~*Standard*) is currently under study~~ by NICE. To avoid any ambiguity or contradiction, all discussions of Kernel Library classes in this book have been replaced by references to the ~~PELKS.~~

CHANGES

This book ~~correspnds~~ to the second printing of *Eiffel: The Language*. It incorporates a few corrections corresponding to mistakes that have been detected since the publication of the book. The principal among these changes are listed in chapter 14 of *ISE Eiffel: The Environment* (Technical Report TR-39/IE).

A complete list of the changes will be made available separately. Time prevented inclusion of change bars for the first release of this document, but they will be part of future editions; all changes have been carefully logged.

In case of discrepancy between *Eiffel: The Language* and *Eiffel: The Reference*, follow the document that has the latest printing date, unless the problem appears to be due to a text processing mistake. Remember that at the basis of Eiffel there is only one document; the only differences result from how one selects and prints portions of that document. This is the required condition for the stability, maturity and credibility of the Eiffel language; in other words, for its success.

The present book uses the same conventions as *Eiffel: The Language*. To avoid any confusion, the original chapter and section numbers have been retained. So if you have leafed through this book before reading the present preface, you may have wondered why the first chapter is number 2 and its first section 2.2; but the purpose is clear: making it easy to find the corresponding place in *Eiffel: The Language* if when examining some part of the edifice you want to retrieve the scaffolding — in other words some of the supporting comments and examples. Using a different section numbering scheme for the present book would have caused endless misunderstandings.

Unlike the sections, the pages have been renumbered, since consecutive numbering facilitates searching for specific information.

In contrast with *Eiffel: The Language*, this book avoids repetition of any kind. This is why you will find no syntax summary or syntax diagram. Such elements (which are present in the complete description) will be easy to add if readers feel they are necessary.

There is currently no index, but this omission will be repaired in a future version.

Finally, please remember that the present edition is only the second release of *Eiffel: The Reference*. In particular, some errors may remain in the software that served to extract the information and produce the index. Indulgence is thus requested from the reader.

ACKNOWLEDGMENTS

In addition to the acknowledgments included at the end ~~of~~ the preface to *Eiffel: The Language*, it is appropriate to thank the various readers who have contributed comments since the book's publication, most remarkably David Hollenberg from the Information Systems Institute of the University of Southern California and Helmut Weber from Austria, formerly from IBM.

THE SHORT FORM

The worst that could happen to the description of Eiffel is to follow the fate of so many earlier languages: the emergence of several descriptions, each slightly incompatible with the others. *Eiffel: The Language* went to considerable length to prevent this from happening. By threading several levels of discourse into a single cloth, the book was able to forestall eventual divergence.

In particular, the book relied on an extensive system of "road signs" to identify each of the interwoven threads: syntax, validity, semantics, comment, caveat, preview, reminder.

As this system is directly reflected in the source electronic form of the original document, it is possible to use software tools, aided by a little human intervention, to extract one or several of the threads. This approach is what made the present book possible. It is not, strictly speaking, a new book, but an **extract** of the relevant parts of *Eiffel: The Language*.

Another way of expressing this observation is to claim that producing this book was, to a certain extent, a software project: writing the tools that would extract the essential parts of the complete document and ignore the rest. This effort required some "massaging", as programmers say, of the original text, to mark some parts as retained and others as discarded. But the extent of that massaging was remarkably limited: the text was organized in such a systematic way that most of the extraction could be done automatically, based on a number of selection rules not unlike those of a little expert systems.

Anyone familiar with the Eiffel method will have recognized the idea: it is the notion of short form. To document an Eiffel class, you do not as a rule write a separate document; you should instead include the relevant information in the class text itself, and rely on computer tools to extract views of the class at various levels of abstraction, in particular the short (or flat-short) form which only keeps the interface properties of the class – signatures, preconditions, postconditions and header comments of exported features, class invariant – while discarding the implementation information (non-exported features, routine bodies, distinction between functions and attributes). Here we are doing the same with respect to the language itself: using the power of the computer to remove the non-essential information from a complete description.

The major advantage, in the case of classes, is that we can keep a single description. So when things evolve – as they inevitably will, be they classes or languages – we have only one document to maintain. This may be called the **principle of single reference** and is essential to the smooth evolution of the language and its description.

Although the work of the NICE library committee may cause changes to be brought to the present document independently of *Eiffel: The Language*, every effort will be made to maintain the principle of single reference, avoiding the disaster that divergence between the two documents would mean for Eiffel. In particular, the structure and section numbering will be kept the same for the two books, and changes made to the present one as a result of the committee's work will be continuously reflected back into the electronic version of *Eiffel: The Language*.

LIBRARY ELEMENTS

The definition of Eiffel relies on a number of predefined classes in ~~the "~~ Kernel ~~Library",~~ covering such fundamental notions as basic types (*BOOLEAN*, *INTEGER* and the like), arrays, input and output.

Preface

This document serves two purposes:

- It is submitted to the Language Committee of the *Nonprofit International Consortium for Eiffel* as the second step towards a standard definition of Eiffel (the first step was the book *Eiffel: The Language*).
- It provides users of Eiffel with a short language definition.

For the first of these goals, note that the present Preface is not part of the intended Standard.

WHAT IS EIFFEL?

The name Eiffel covers a method and a language for the systematic development of quality software, based on the full application of object-oriented principles.

Only the language aspect is covered in this book. Other documents are available on the method and on the various compilers, tools and environments that make it possible to develop software with Eiffel.

A complete description of the Eiffel language has been previously published: the book *Eiffel: The Language* (Prentice Hall, second printing with corrections, 1992, ISBN 0-13-24795-7). The intent of that document is to provide under a single cover a precise reference, a tutorial, a Guide for the Perplexed, and a detailed user's manual for the language. As a result, much of the space in it is occupied by examples, explanations, justifications, discussions, previews, reminders and comments. *Eiffel: The Language* does not shun repetition; occasionally, for example, some part of ~~the~~ discussion needs to refer to the syntax of a construct seen in a distant chapter, and simply reproduces the syntax specification for the convenience of the reader.

In some cases, however, a shorter reference may be necessary. A typical example is that of someone implementing an Eiffel compiler, who may be presumed to be familiar with the rationale behind the various components of the language, but will need a document ~~were~~ he can quickly find precise answers to specific questions, often on fine points ("Can the target of an anchored declaration be anchored too?"). Another example is that of a user who is familiar with the language but wants to keep a concise reference on his desk.

Providing such a no-frills description of Eiffel is the purpose of *Eiffel: The Reference*.

> This book is not meant as a first introduction to Eiffel. If you do not know the language, or know it only superficially, you should read *Eiffel: The Language*. The present book will mostly be useful as a summary of *Eiffel: The Language* for readers who are familiar with that earlier book.

Must it be assumed that because we are engineers beauty is not our concern, and that while we make our constructions robust and durable we do not also strive to make them elegant?

Is it not true that the genuine conditions of strength always comply with the secret conditions of harmony?

The first principle of architectural esthetics is that the essential lines of a monument must be determined by a perfect adaptation to its purpose.

Gustave Eiffel, 1887

From his response in the newspaper *Le Temps t*o a petition by members of the literary and artistic Establishment protesting his project of elevating a tower of iron in Paris.

Book identification

Eiffel: The Reference, ISE Technical Report TR-EI-41/ER.

Publication history

First published as *Eiffel: The Language* (TR-EI-2/BR) in 1988. Successive versions appeared in subsequent years and were replaced by TR-EI-17/RM, *Eiffel: The Language* in 1991, second revised printing 1992, also available as a book published by Prentice Hall, ISBN 0-13-245-925-~~7. First edition of *Eiffel: The Reference* under the present report number appeared in 1992. The present revision is 3.3.4, corresponding to version 3.3.4 of the ISE Eiffel environment. It is submitted to the Language Committee of the Nonprofit International Consortium for Eiffel as revision 2 of the proposed Eiffel language standard.~~

The majority of the material in the present book is excerpted from *Eiffel: The Language*.

Author

Bertrand Meyer.

Software credits

See "Credits" at the end of the Preface and in *Eiffel: The Language.*

Cover design

Rich Ayling.

Copyright notice and proprietary information

Copyright © Bertrand Meyer, 1992, 1995.

The material from *Eiffel: The Language* is reproduced here under permission from copyright holder and the publisher for the exclusive benefit of users of ISE Eiffel 3. In addition, the copyright holder grants permission to members in good standing of the Nonprofit International Consortium for Eiffel to make any use of this document that does not infringe on the rights of other parties, in particular the publisher.

Eiffel:

The Reference